

# **Programa de teoría**

## **AED I. Estructuras de Datos.**

### **→ 1. Abstracciones y especificaciones.**

2. Conjuntos y diccionarios.
3. Representación de conjuntos mediante árboles.
4. Grafos.

## **AED II. Algorítmica.**

1. Análisis de algoritmos.
2. Divide y vencerás.
3. Algoritmos voraces.
4. Programación dinámica.
5. Backtracking.
6. Ramificación y poda.

# **AED I: ESTRUCTURAS DE DATOS**

## **Tema 1. Abstracciones y Especificaciones.**

- 1.1. Abstracciones, tipos y mecanismos.
- 1.2. Especificaciones informales.
- 1.3. Especificaciones formales.
  - 1.3.1. Método axiomático (o algebraico).
  - 1.3.2. Método constructivo (u operacional).

# 1.1. Abstracciones, tipos y mecanismos.

```
290 DIM b$(22,2): FOR n=1 TO 1: FOR m=1 TO 2
300 LET s=INT (RND*22)+1
310 IF b$(s,1)=" " THEN LET b$(s,1)=a$(n,1): LET b$(s,2)=a$(n,2):
NEXT m: NEXT n: GO TO 330
320 GO TO 300
330 DIM f(22): LET di=0: LET itn=0: LET u=.001
340 PRINT AT 20,2;di: IF di=275000 THEN LET di=350000: PRINT AT
20,2; FLASH 1;di;"CONSEGUIDO EL PLENO EN ";itn;" veces": PRINT
#0;"Pulsa una tecla para empezar": GO SUB 440: GO SUB 440: GO SUB
440: PAUSE 0: GO TO 350
350 INPUT n: IF n>22 OR n<1 THEN GO TO 350
360 IF r(n)=1 THEN GO TO 350
370 LET k=n: GO SUB 700
380 INPUT m: IF m>22 OR m<1 OR m=n THEN GO TO 380
390 IF r(m)=1 THEN GO TO 380
400 LET k=m: GO SUB 700
410 LET itn=itn+1: IF b$(n)=b$(m) THEN LET di=di+250000: PAPER 3: LET
k=n: GO SUB 720: PAPER 3: LET k=m: GO SUB 720: LET r(n)=1: LET
r(m)=1: GO SUB 440: GO SUB 450: GO TO 340
420 BRIGHT 1: PAUSE 45: PAUSE 45: LET f=f(n): LET c=c(n): PRINT AT
f,c;a$(n,8);AT f+1,c;a$(n,14);AT f+2,c;a$(n,20): PRINT AT
f,c;a$(n,7 TO 8);AT f+1,c;a$(n,13 TO 14);AT f+2,c;a$(n,19 TO 20):
BEEP .01,-10: PRINT a$(n): BEEP .02,0
430 LET f=f(m): LET c=c(m): PRINT AT f,c;a$(m,8);AT
f+1,c;a$(m,14);AT f+2,c;a$(m,20): PRINT AT f,c;a$(m,7 TO 8);AT
f+1,c;a$(m,13 TO 14);AT f+2,c;a$(m,19 TO 20): BEEP .01,-10: PRINT
a$(m): BEEP .02,0: BRIGHT 0: GO TO 350
```

→ **procedure** ordenar(a: array;  
var b: array);

→ **type** persona;

→ **class** pila;

**Abstraer:** Eliminar lo irrelevante y quedarnos con lo realmente importante.

¿Qué es lo importante?

# **1.1. Abstracciones, tipos y mecanismos.**

## **MECANISMOS DE ABSTRACCIÓN**

**Abstracción por especificación:** Solo necesitamos conocer qué hace un procedimiento y no cómo funciona.  
(Encapsulación y ocultación de implement.)

**Abstracción por parametrización:** Un algoritmo, un tipo, o una variable se definen en base a unos parámetros.  
(Genericidad)

# 1.1. Abstracciones, tipos y mecanismos.

## TIPOS DE ABSTRACCIONES

- Abstracciones funcionales → Rutinas, funciones, procedimientos
- Abstracciones de datos → Tipos Abstractos de Datos (TAD)
- Abstracciones de iteradores → Iteradores

## **1.1. Abstracciones, tipos y mecanismos.**

### **TIPO ABSTRACTO DE DATOS:**

Dominio abstracto de valores junto con las operaciones aplicables sobre el mismo.

### **TIPO DE DATOS:**

Conjunto de valores que puede tomar una variable, un parámetro o una expresión.

### **ESTRUCTURA DE DATOS:**

Disposición en memoria de los datos necesarios para almacenar valores de un tipo.

# 1.1. Abstracciones, tipos y mecanismos.

## Ejemplo:

- **TAD:** Entero,  $\mathbb{Z}$
- **Tipo de datos:** Tipo `int` de C/C++ o el tipo `Integer` de Java
- **Estructura de datos:** Representación mediante enteros de 32 bits, 64 bits, listas de dígitos (enteros largos), etc.

# 1.1. Abstracciones, tipos y mecanismos

- La evolución de los lenguajes de programación tiende a introducir cada vez más abstracciones.

Lenguajes  
de bajo nivel

Lenguajes  
estructurados

Lenguajes  
orientados a objetos



(Basic, Fortran,  
Ensamblador, ...)

(Pascal, C,  
Modula, ADA, ...)

(Smalltalk, C++,  
Java, Python, ...)



# 1.1. Abstracciones, tipos y mecanismos

- La evolución de los lenguajes de programación tiende a introducir cada vez más abstracciones.
- Soporte de TAD:
  - **Lenguajes estructurados (tipos definidos por el usuario):** Los datos y las operaciones van aparte.
  - **Lenguajes orientados a objetos (clases):** Los datos y las operaciones constituyen una unidad, el concepto de clase.

# 1.1. Abstracciones, tipos y mecanismos.

## C

```
struct Pila {  
    int tope;  
    int datos[10];  
};  
  
void push (Pila *p, int valor);  
void pop (Pila *p);  
int top (Pila p);
```

## C++

```
class Pila {  
    private:  
        int tope;  
        int datos[10];  
    public:  
        void push (int valor);  
        void pop ( );  
        int top ( );  
};
```

## Pila

datos

tope  
→

0	34
1	20
2	51
3	
4	
5	
6	
7	
8	
9	

# 1.1. Abstracciones, tipos y mecanismos.

## C

```
Pila p1, p2;  
int i;  
  
push(&p1, 34);  
push(&p1, 20);  
push(&p1, 51);  
pop(&p1);  
i= top(p1);  
p1.tope= 243;  
i= top(p1);  
...
```

**Error de ejecución:**  
se sale del array datos

## C++

```
Pila p1, p2;  
int i;  
  
p1.push(34);  
p1.push(20);  
p1.push(51);  
p1.pop();  
i= p1.top();  
p1.tope= 243;  
i= p1.top();  
...
```

**Error de compilación:**  
tope es privado

Pila  
datos

tope  
→

0	34
1	20
2	51
3	
4	
5	
6	
7	

# **1.1. Abstracciones, tipos y mecanismos.**

## **Especificaciones: Tipos de notaciones**

- Notaciones informales.
- Notaciones formales.
  - Algebraicas (o axiomáticas).
  - Operacionales (o constructivas).

# 1.2. Especificaciones informales.

## 1.2.1. Abstracciones funcionales.

### Notación

**Operación** <nombre> (**ent** <id>: <tipo>; <id>: <tipo>, ..., **sal** <tipo>)

**Requiere:** Establece restricciones de uso.

**Modifica:** Identifica los datos de entrada que se modifican (si existe alguno).

**Calcula:** Descripción textual del comportamiento de la operación.

# 1.2. Especificaciones informales.

## 1.2.1. Abstracciones funcionales.

**Ejemplo 1:** Eliminar la repetición en los elementos de un array.

**Operación** QuitarDuplic (**ent**  $a$ : array [entero])

**Modifica:**  $a$

**Calcula:** Quita los elementos repetidos de  $a$ . El límite inferior del array no varía, pero sí lo puede hacer el superior.

**Ejemplo 2:** Buscar un elemento en un array de enteros.

**Operación** Buscar (**ent**  $a$ : array [entero];  $x$ : entero; **sal**  $i$ : entero)

**Requiere:**  $a$  debe estar ordenado de forma ascendente.

**Calcula:** Si  $x$  está en  $a$ , entonces  $i$  debe contener el valor del índice de  $x$  tal que  $a[i] = x$ . Si  $x$  no está en  $a$ , entonces  $i = \text{sup} + 1$ , donde  $\text{sup}$  es el índice superior del array  $a$ .

## 1.2.1. Abstracciones funcionales.

**Generalización:** una operación está definida independientemente de cuál sea el tipo de sus parámetros.

**Ejemplo 3:** Eliminar la repetición en los elementos de un array.

**Operación** QuitarDuplic [T: tipo](**ent**  $a$ : array [T])

**Requiere:** T debe tener una operación de comparación IgualQue(ent T, T; sal booleano).

**Modifica:**  $a$

**Calcula:** Quita los elementos repetidos de  $a$ . El límite inferior del array no varía, pero sí lo puede hacer el superior.

**Ejemplo 4:** Buscar un elemento en un array de enteros.

**Operación** Buscar [T: tipo](**ent**  $a$ : array [T];  $x$ : T; **sal**  $i$ : entero)

**Requiere:** T debe tener dos operaciones de comparación MenorQue(ent T, T; sal bool), Igual(ent T, T; sal bool).  
 $a$  debe estar ordenado de forma ascendente.

**Calcula:** Si  $x$  está en  $a$ , entonces  $i$  debe contener ...

# 1.2.1. Abstracciones funcionales.

- Ejemplo de especificación informal de funciones:  
Especificación de las librerías STL de C++: [www.cplusplus.com](http://www.cplusplus.com)

Nombre de la operación	→	<b>std::list::push_back</b> <span style="float: right;">&lt;list&gt;</span>
Sintaxis	→	<div>C++98 C++11 ?</div> <div><code>void push_back (const value_type&amp; val);</code></div>
Descripción en lenguaje natural	→	<b>Add element at the end</b> Adds a new element at the end of the <code>list</code> container, after its current last element. The content of <code>val</code> is copied (or moved) to the new element.  This effectively increases the container <code>size</code> by one.
Parámetros	→	<b>Parameters</b> <div>val</div> <div>Value to be copied (or moved) to the new element. Member type <code>value_type</code> is the type of the elements in the container, defined in <code>list</code> as an alias of its first template parameter (<code>T</code>).</div>
Valor devuelto	→	<b>Return value</b> none  The storage for the new elements is allocated using the container's <code>allocator</code> , which may throw exceptions on failure (for the default <code>allocator</code> , <code>bad_alloc</code> is thrown if the allocation request does not succeed).
Ejemplo de uso	→	<b>Example</b> <pre>1 // list::push_back 2 #include &lt;iostream&gt; 3 #include &lt;list&gt; . 19 return 0; 20 }</pre>
Orden de complejidad	→	<b>Complexity</b> Constant.



# **1.2. Especificaciones informales.**

## **1.2.2. Abstracciones de datos.**

### **Notación**

**TAD** <nombre\_tipo> **es** <lista\_operaciones>

#### **Descripción**

Descripción textual del tipo

#### **Operaciones**

Especificación informal de las operaciones de la lista anterior

**Fin** <nombre\_tipo>.

## 1.2.2. Abstracciones de datos.

**TAD** CjtoEnteros **es** Vacío, Insertar, Suprimir, Miembro, EsVacío, Unión, Intersección, Cardinalidad

### Descripción

Los CjtoEnteros son conjuntos matemáticos modificables, que almacenan valores enteros.

### Operaciones

**Operación** Vacío (**sal** CjtoEnteros)

**Calcula:** Devuelve un conjunto de enteros vacío.

**Operación** Insertar (**ent** c: CjtoEnteros; x: entero)

**Modifica:** c.

**Calcula:** Añade x a los elementos de c. Después de la inserción, el nuevo conjunto es  $c \cup \{x\}$ .

...

**Fin** CjtoEnteros.

## 1.2.2. Abstracciones de datos.

**TAD** ListaEnteros **es** Crear, Insertar, Primero, Ultimo, Cabeza, Cola, EsVacio, Igual

### Descripción

Las ListaEnteros son listas de enteros modificables. Las listas se crean con las operaciones Crear e Insertar...

### Operaciones

**Operación** Crear (**sal** ListaEnteros)

**Calcula:** Devuelve una lista de enteros vacía.

**Operación** Insertar (**ent**  $\vdash$  ListaEnteros; x: entero)

**Modifica:** /.

**Calcula:** Añade x a la lista / en la primera posición.

...

**Fin** ListaEnteros.

## 1.2.2. Abstracciones de datos.

- **Generalización (parametrización de tipo):** El tipo se define en función de otro tipo pasado como parámetro.
- Útil para definir tipos **contenedores** o **colecciones**. P. ej. Listas, pilas, colas, arrays, conjuntos, etc.

- En lugar de:

ListaEnteros  
ListaCadenas  
ListaReales  
....



- Tenemos:  
Lista[T]

- Instanciación: Lista[entero], Lista[cadena],...

## 1.2.2. Abstracciones de datos.

**TAD** Conjunto[T: tipo] **es** Vacío, Insertar, Suprimir, Miembro, EsVacío, Unión, Intersección, Cardinalidad

### Descripción

Los Conjunto[T] son conjuntos matemáticos modificables, que almacenan valores de tipo T.

### Operaciones

**Operación** Vacío (**sal** Conjunto[T])

...

**Operación** Insertar (**ent** c: Conjunto[T]; x: T)

...

**Operación** Suprimir (**ent** c: Conjunto[T]; x: T)

...

**Operación** Miembro (**ent** c: Conjunto[T]; x: T; **sal** booleano)

...

**Fin** Conjunto.

## 1.2.2. Abstracciones de datos.

**TAD** Lista[T] **es** Crear, Insertar, Primero, Ultimo, Cabeza, Cola, EsVacío, EsIgual

### Descripción

Las Lista[T] son listas modificables de valores de tipo T.  
Las listas se crean con las operaciones Crear e Insertar...

### Operaciones

**Operación** Crear (**sal** Lista[T])

...

**Operación** Insertar (**ent** /: Lista[T]; x: T)

...

**Operación** Primero (**ent** /: Lista[T]; **sal** T)

...

**Fin** Lista.

## 1.2.2. Abstracciones de datos.

- En C++ es posible definir tipos parametrizados.
- Plantillas **template**:

```
template <class T>
class Pila {
    private:
        int tope;
        int maxDatos;
        T *datos;
    public:
        Pila (int maximo = 10);
        void Push (T valor);
        T Pop ();
};
```

- Instanciación:

```
Pila<int> p1; Pila<string> p2;
```

# 1.2. Especificaciones informales.

## 1.2.3. Abstracciones de iteradores.

- **Ejemplo:** Sobre el TAD CjtoEnteros queremos añadir operaciones para calcular la suma, el producto, ...

**Operación** suma\_conj (**ent** c: CjtoEnteros; **sal** entero)

**Calcula:** Devuelve la suma de los elementos de c.

....

**Operación** producto\_conj (**ent** c: CjtoEnteros; **sal** entero)

....

**Operación** varianza\_conj (**ent** c: CjtoEnteros; **sal** real)

....



## 1.2.3. Abstracciones de iteradores.

- Necesitamos abstracciones de la forma:
  - para cada** elemento  $i$  del conjunto  $A$   
**hacer** acción sobre  $i$
  - para cada** elemento  $i$  de la lista  $L$   
**hacer** acción sobre  $i$
  - para cada**  $i$  de la cola  $C$  **tal que**  $P(i)$   
**hacer** acción sobre  $i$
  - $D :=$  **Seleccionar** todos los  $i$  de  $C$  **tal que**  $P(i)$
- Abstracción de **iteradores**: permiten definir un recorrido abstracto sobre los elementos de una colección.

## **1.2.3. Abstracciones de iteradores.**

- La abstracción de iteradores no es soportada por la mayoría de los lenguajes de programación estructurados.
- En los lenguajes más modernos, se implementa con mecanismos auxiliares.
- **Posibles definiciones de los iteradores:**
  - Como una abstracción funcional: operación/instrucción de iteración.
  - Como una abstracción de datos: tipo de datos iterador.

## 1.2.3. Abstracciones de iteradores.

- **Iterador como una abstracción funcional:**

**Iterador** ParaTodoHacer [T: tipo] (**ent** C: Conjunto[T]; *acción*: Operación)

**Requiere:** *acción* debe ser una operación que recibe un parámetro de tipo T y no devuelve nada, *acción(ent T)*.

**Calcula:** Recorre todos los elementos *c* del conjunto C, aplicando sobre ellos la operación *acción(c)*.

- Ejemplos de abstracción funcional de iteradores en C++ (STL):

<code>list&lt;int&gt; lista;</code>	<code>queue&lt;string&gt; cola;</code>	<code>string cadena;</code>
<code>...</code>	<code>...</code>	<code>...</code>
<code>for (int i: lista)</code>	<code>for (auto e: cola)</code>	<code>for (auto c: cadena)</code>
<code>{</code>	<code>{</code>	<code>{</code>
<code>    // Procesar i</code>	<code>    // Procesar e</code>	<code>    // Procesar c</code>
<code>}</code>	<code>}</code>	<code>}</code>

## 1.2.3. Abstracciones de iteradores.

- **Iterador como una abstracción de datos:**

**Tipoliterador** IteradorPreorden [T: tipo] **es** Iniciar, Actual, Avanzar, EsFinal

### **Descripción**

Los valores de tipo IteradorPreorden[T] son iteradores definidos sobre árboles binarios de cualquier tipo T. Los elementos del árbol son devueltos en preorden. El iterador se debe inicializar con Iniciar.

### **Operaciones**

**Operación Iniciar** (**ent** A: ArbolBinario[T]; **sal** IteradorPreorden)

**Calcula:** Devuelve un iterador nuevo, colocado sobre la raíz de A.

**Operación Actual** (**ent** *iter*: IteradorPreorden; **sal** T)

...

**Fin** IteradorPreorden.

## 1.2.3. Abstracciones de iteradores.

```
var
  A: ArbolBinario[T];
  i: IteradorPreorden[T];
begin
  ...
  i:= Iniciar(A);
  while Not EsFinal(i) do begin
    Acción sobre Actual(i);
    i:= Avanzar(i);
  end;
  ...
```

Recordatorio de I.P.:  
Primer modelo de  
acceso secuencial

## 1.2.3. Abstracciones de iteradores.

- Ejemplos de abstracción de iteradores mediante un tipo de datos iterador en C++ (STL):

```
list<int> lista;  
...  
list<int>::iterator it;  
it= lista.begin();  
while (it!=lista.end()) {  
    // Acción sobre *it  
    it++;  
}
```

Iniciar  
EsFinal  
Actual  
Avanzar

```
list<int> lista;  
...  
for (auto it= lista.begin(); it!=lista.end(); it++) {  
    // Acción sobre *it  
}
```

# 1.3. Especificaciones formales.

- Las especificaciones en lenguaje natural son ambiguas e imprecisas.
- **Especificaciones formales:** definen un TAD o una operación de manera precisa, utilizando un lenguaje matemático.
- Ventajas de una especificación formal:
  - **Prototipado.** Las especificaciones formales pueden llegar a ser ejecutables.
  - **Corrección del programa.** Verificación automática y formal de que el programa funciona correctamente.
  - **Reusabilidad.** Posibilidad de usar la especificación formal en distintos ámbitos.

# 1.3. Especificaciones formales.

## Notación

La descripción formal constará de cuatro partes:

- **NOMBRE.** Nombre genérico del TAD.
- **CONJUNTOS.** Conjuntos de datos que intervienen en la definición.
- **SINTAXIS.** Signatura de las operaciones definidas.
- **SEMÁNTICA.** Indica el significado de las operaciones, cuál es su resultado.



# 1.3. Especificaciones formales.

- **Sintaxis:**

$\langle \text{nombre\_operación} \rangle : \langle \text{conj\_dominio} \rangle \rightarrow \langle \text{conj\_resultado} \rangle$

- Las distintas notaciones formales difieren en la forma de definir la semántica:
  - **Método axiomático o algebraico.** Se establece el significado de las operaciones a través de relaciones entre operaciones (*axiomas*). Significado implícito de las operaciones.
  - **Método constructivo u operacional.** Se define cada operación por sí misma, independientemente de las otras, basándose en un modelo subyacente. Significado explícito de las operaciones.

### 1.3.1. Método axiomático (o algebraico).

- La semántica de las operaciones se define a través de un conjunto de **axiomas**.
- Un axioma es una regla que establece la igualdad de dos expresiones:

$$<\text{Expresión 1}> = <\text{Expresión 2}>$$

- Por ejemplo:

Suma (dos, dos) = Producto (dos, dos)

Tope (Push (x, pila)) = x

OR (verdadero, b) = verdadero

### 1.3.1. Método axiomático (o algebraico).

- ¿Qué axiomas introducir en la semántica?
- Los axiomas deben ser los necesarios para satisfacer dos propiedades:
  - **Completitud:** Los axiomas deben ser los suficientes para poder deducir el significado de cualquier expresión.
  - **Corrección:** A partir de una expresión solo se puede obtener un resultado.

### 1.3.1. Método axiomático (o algebraico).

- **Ejemplo:** TAD Natural de los números naturales.

# NOMBRE

# Natural

# CONJUNTOS

N	Conjunto de naturales
---	-----------------------

Bool    Conjunto de booleanos {true, false}

# SINTAXIS

cero:  $\rightarrow$  N

sucesor:  $N \rightarrow N$

suma:  $N \times N \rightarrow N$

esCero:            N             $\rightarrow$     Bool

**esIguar:**      **N x N**    **→**      **Bool**

## 1.3.1. Método axiomático (o algebraico).

### SEMÁNTICA

$\forall m, n \in \mathbb{N}$

1.  $\text{suma}(\text{cero}, n) = n$

2.  $\text{suma}(\text{sucesor}(m), n) = \text{sucesor}(\text{suma}(m, n))$

3.  $\text{esCero}(\text{cero}) = \text{true}$

4.  $\text{esCero}(\text{sucesor}(n)) = \text{false}$

5.  $\text{esIgual}(\text{cero}, n) = \text{esCero}(n)$

6.  $\text{esIgual}(\text{sucesor}(n), \text{cero}) = \text{false}$

7.  $\text{esIgual}(\text{sucesor}(n), \text{sucesor}(m)) = \text{esIgual}(n, m)$

### 1.3.1. Método axiomático (o algebraico).

- **Ejecución de una especificación algebraica:**  
aplicar sucesivamente las reglas de la semántica hasta que no se puedan aplicar más.
- **Ejemplo.** ¿Cuánto valen las siguientes expresiones?
  - a) suma (suma(sucesor(cero), cero), sucesor (cero) )
  - b) esIgual (sucesor (sucesor (cero)), suma (suma (sucesor (cero), cero), sucesor (cero) ) )

## 1.3.1. Método axiomático (o algebraico).

- Supongamos un TAD,  $T$ .
- **Tipos de operaciones:**
  - **Constructores.** Conjunto mínimo de operaciones del TAD, a partir del cual se puede obtener cualquier valor del tipo  $T$ .
$$\underline{c1}: \rightarrow T, \underline{c2}: V \rightarrow T, \underline{c3}: V_1x...xV_n \rightarrow T$$
  - **Modificación.** A partir de un valor del tipo, obtienen otro valor del tipo  $T$ , y no son constructores.
$$\underline{m1}: T \rightarrow T, \underline{m2}: TxV \rightarrow T, \underline{m3}: V_1x...xV_n \rightarrow T$$
  - **Consulta.** Devuelven un valor que no es del tipo  $T$ .
$$\underline{o1}: T \rightarrow V, \underline{o2}: TxV \rightarrow V', \underline{o3}: V_1x...xV_n \rightarrow V_{n+1}$$

### **1.3.1. Método axiomático (o algebraico).**

- La ejecución de una expresión acaba al expresarla en función de los constructores.
- ¿Cómo garantizar que una especificación es completa y correcta?
- Definir los axiomas suficientes para relacionar las operaciones de modificación y consulta con los constructores.
- No incluir axiomas que se puedan deducir de otros existentes.



## 1.3.1. Método axiomático (o algebraico).

- **Ejemplo:** Especificación del TAD genérico pila.

### NOMBRE

Pila [T]

### CONJUNTOS

P      Conjunto de pilas

T      Conjunto de elementos que pueden ser almacenados

Bool   Conjunto de booleanos {true, false}

### SINTAXIS

pilaVacía:                       $\rightarrow$       P

esVacía:              P    $\rightarrow$       Bool

pop:                      P    $\rightarrow$       P

tope:                      P    $\rightarrow$       T

push:              T x P  $\rightarrow$       P

### 1.3.1. Método axiomático (o algebraico).

- En el caso de **tope**:  $P \rightarrow T$ , ¿qué pasa si la pila está vacía?
- Se puede añadir un conjunto de mensajes en **CONJUNTOS**, de la forma:

M      Conjunto de mensajes {"Error. La pila está vacía"}

- Y cambiar en la parte de **SINTAXIS** la operación **tope**:

tope:  $P \rightarrow T \cup M$

## 1.3.1. Método axiomático (o algebraico).

	pilaVacía	push (t, p)
esVacía ( )	esVacía(pilaVacía) =	esVacía(push(t, p)) =
pop ( )	pop(pilaVacía) =	pop(push(t, p)) =
tope ( )	tope(pilaVacía) =	tope(push(t, p)) =

## 1.3.1. Método axiomático (o algebraico).

### SEMÁNTICA

$\forall t \in T; \forall p \in P$

1. esVacía (pilaVacía) = true

2. esVacía (push (t, p)) = false

3. pop (pilaVacía) = pilaVacía

4. pop (push (t, p)) = p

5. tope (pilaVacía) = “Error. La pila está vacía”

6. tope (push (t, p)) = t

### 1.3.1. Método axiomático (o algebraico).

- Calcular:
  - a) `pop(push(3, push(2, pop(pilaVacía))))`
  - b) `tope(pop(push(1, push(2, pilaVacía))))`
- Añadir una operación **esIgual** para comparar dos pilas.
- ¿Cómo hacer que la operación **pop** devuelva el tope de la pila y al mismo tiempo lo saque de la pila?

## 1.3.1. Método axiomático (o algebraico).

- Para facilitar la escritura de la expresión del resultado en la semántica, se pueden emplear condicionales de la forma:

**SI** <condición>  $\Rightarrow$  <valor si cierto> | <valor si falso>

- **Ejemplo:** Especificación algebraica del TAD bolsa.

### NOMBRE

Bolsa[T]

### CONJUNTOS

B	Conjunto de bolsas
T	Conjunto de elementos
Bool	Conjunto de booleanos {true, false}
N	Conjunto de naturales

### SINTAXIS

bolsaVacía:	$\rightarrow$	B
poner:	$T \times B \rightarrow$	B
esVacía:	$B \rightarrow$	Bool
cuántos:	$T \times B \rightarrow$	N

### **1.3.1. Método axiomático (o algebraico).**

- Incluir una operación quitar, que saque un elemento dado de la bolsa.
- ¿Y si queremos que los saque todos?
- Incluir una operación esIgual, de comparación de bolsas.

## 1.3.1. Método axiomático (o algebraico).

### Conclusiones:

- Las operaciones no se describen de manera explícita, sino **implícitamente** relacionando el resultado de unas con otras.
- La construcción de los axiomas se basa en un **razonamiento inductivo**.
- ¿Cómo se podría especificar, por ejemplo, un procedimiento de ordenación?



## 1.3.2. Método constructivo (operacional).

- Para cada operación, se establecen las precondiciones y las postcondiciones.
- **Precondición:** Relación que se debe cumplir con los datos de entrada para que la operación se pueda aplicar.
- **Postcondición:** Relaciones que se cumplen después de ejecutar la operación.

## 1.3.2. Método constructivo (operacional). Notación

En la semántica, para cada operación **<nombre>**:

**pre-<nombre>**(**<param\_entrada>**) ::= **<condición\_lógica>**

**post-<nombre>**(**<param\_entrada>**; **<param\_salida>**) ::= **<condición\_lógica>**

- **Ejemplo:** operación **máximo**, que tiene como entrada dos números reales y da como salida el mayor de los dos.

**máximo:**  $R \times R \rightarrow R$  *(SINTAXIS)*

**pre-máximo**(x, y) ::= true *(SEMANTICA)*

**post-máximo**(x, y; r) ::=  $(r \geq x) \wedge (r \geq y) \wedge (r=x \vee r=y)$

## 1.3.2. Método constructivo (operacional).

- **Ejemplo:** operación **máximo** sobre números reales, pero restringida a números positivos.

**máximop:**  $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$

**pre-máximop**( $x, y$ ) ::=  $(x \geq 0) \wedge (y \geq 0)$

**post-máximop**( $x, y; r$ ) ::=  $(r \geq x) \wedge (r \geq y) \wedge (r=x \vee r=y)$

- ¿Qué sucedería si  $x$  o  $y$  no son mayores que 0?
- No se cumple la precondition  $\rightarrow$  No podemos asegurar que se cumpla la postcondición.

## 1.3.2. Método constructivo (operacional).

- Implementación en C/C++ de pre- y post-condic.

máximop:  $R \times R \rightarrow R$

**pre-máximop**(x, y) ::=  $(x \geq 0) \wedge (y \geq 0)$

**post-máximop**(x, y; r) ::=  $(r \geq x) \wedge (r \geq y) \wedge (r=x \vee r=y)$

```
double maximop (double x, double y)
{
    assert(x>=0 && y>=0); // Precondición
    double r;
    if (x>y) r= x; else r= y;
    assert(r>=x && r>=y && (r==x || r==y)); //Postcondic
    return r;
}
```

## 1.3.2. Método constructivo (operacional).

- **Otra posibilidad:** Definir un conjunto **M** (de mensajes de error) y cambiar la imagen. Modificar la sintaxis y la semántica:

**máximop2:  $R \times R \rightarrow R \cup M$**

**pre-máximop2**(x, y) ::= true

**post-máximop2**(x, y; r) ::=  $SI (x \geq 0) \wedge (y \geq 0)$

$\Rightarrow (r \geq x) \wedge (r \geq y) \wedge (r=x \vee r=y)$

|  $r = \text{"Fuera de rango"}$

- ¿Cuál es la mejor opción?

## 1.3.2. Método constructivo (operacional).

- ¿Cuál es la mejor solución?
- **La especificación como un contrato.**
- **Contrato de una operación:**  
Si se cumplen unas condiciones en los parámetros de entrada, entonces garantiza una obtención correcta del resultado.



## 1.3.2. Método constructivo (operacional).

- Dos puntos de vista del contrato (especificación):
  - **Implementador. Obligación:** Cumplir la postcondición.  
**Derechos:** Sabe que se cumple la precondition.
  - **Usuario. Obligación:** Cumplir la precondition.  
**Derechos:** Sabe que se cumple la postcondición.
- **Idea:**
  - La operación no trata todos los casos de error, sino que hace uso de las condiciones.
  - La responsabilidad de comprobar la precondition es del que usa la operación.

## 1.3.2. Método constructivo (operacional).

- ¿Cómo se pueden definir las pre/post-condiciones cuando el TAD es más complejo? Por ejemplo, para TAD colecciones.
- Necesitamos un **modelo subyacente**, en el cual se base la definición del TAD.
- No siempre se encuentra uno adecuado...
- **Ejemplo:** Para definir el TAD **Pila[T]**, definiremos el TAD **Lista[T]** por el método axiomático, y luego lo usaremos para definir el TAD pila con el método constructivo.



# 1.3.2. Método constructivo (operacional).

## NOMBRE

Lista[T]

## CONJUNTOS

L Conjunto de listas

T Conjunto de elementos

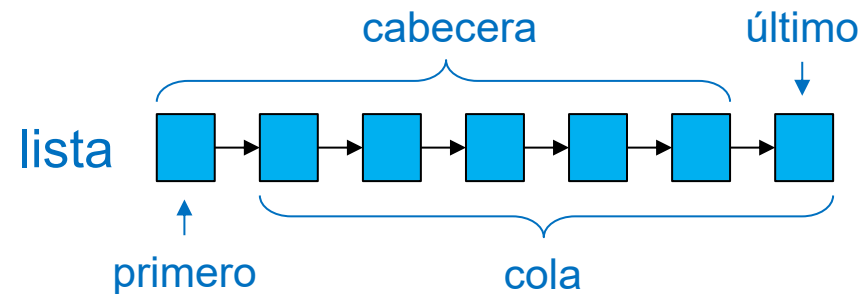
Bool Conjunto de booleanos {true, false}

N Conjunto de naturales

M Conjunto de mensajes {"La lista está vacía"}

## SINTAXIS

crearLista:		→	L
formarLista:	T	→	L
concatenar:	L x L	→	L
último:	L	→	T U M
cabecera:	L	→	L
primero:	L	→	T U M
cola:	L	→	L
longitud:	L	→	N
esListaVacía:	L	→	Bool



# 1.3.2. Método constructivo (operacional).

## SEMÁNTICA

$\forall t \in T; \forall a, b \in L$

1. último (crearLista) = “La lista está vacía”
2. último (formarLista (t)) = t
3. último (concatenar (a, b)) = SI esListaVacía (b)  $\Rightarrow$   
último (a) | último (b)
4. cabecera (crearLista) = crearLista
5. cabecera (formarLista (t)) = crearLista
6. cabecera (concatenar (a, b)) = SI esListaVacía (b)  $\Rightarrow$   
cabecera (a) | concatenar (a, cabecera (b))
7. primero (crearLista) = “La lista está vacía”
8. primero (formarLista (t)) = t
9. primero (concatenar (a, b)) = SI esListaVacía (a)  $\Rightarrow$   
primero (b) | primero (a)

## 1.3.2. Método constructivo (operacional).

- 10. cola (crearLista) = crearLista
- 11. cola (formarLista (t)) = crearLista
- 12. cola (concatenar (a, b)) = SI esListaVacía (a)  $\Rightarrow$   
cola (b) | concatenar (cola (a), b)
- 13. longitud (crearLista) = cero
- 14. longitud (formarLista (t)) = sucesor (cero)
- 15. longitud (concatenar (a, b)) = suma (longitud (a),  
longitud (b))
- 16. esListaVacía (crearLista) = true
- 17. esListavacía (formarLista (t)) = false
- 18. esListaVacía (concatenar (a, b)) = esListaVacía (a)  
AND esListaVacía(b)

## 1.3.2. Método constructivo (operacional).

- Seguimos el ejemplo y aplicamos el método constructivo a la definición de **Pila[T]**, teniendo como modelo subyacente el tipo **Lista[T]**.

### NOMBRE

Pila[T]

### CONJUNTOS

S Conjunto de pilas

T Conjunto de elementos

Bool Conjunto de valores booleanos {true, false}

M Conjunto de mensajes {"La pila está vacía"}

### SINTAXIS

crearPila:  $\rightarrow$  S

tope: S  $\rightarrow$  T U M

pop: S  $\rightarrow$  S U M

push: T x S  $\rightarrow$  S

esVacíaPila: S  $\rightarrow$  Bool

A.E.D. I

# 1.3.2. Método constructivo (operacional).

## SEMÁNTICA

$\forall t \in T; \forall s, r \in S; b \in \text{Bool}; t' \in T \cup M; p \in S \cup M$

1. pre-crearPila () ::= true

2. post-crearPila () ::= true

3. pre-tope (s) ::= true

4. post-tope (s; t') ::= Si esListaVacía (s)  
 $\Rightarrow t' = \text{"La pila es vacía"}$   
|  $t' = \text{primero (s)}$

5. pre-pop (s) ::= true

6. post-pop (s; p) ::= Si esListaVacía (s)  
 $\Rightarrow p = \text{"La pila está vacía"}$   
|  $p = \text{cola (s)}$

7. pre-push (t, s) ::= true

8. post-push (t, s; r) ::=  $r = \text{concatenar (formarLista (t), s)}$

9. pre-esVacíaPila (s) ::= true

10. post-esVacíaPila (s; b) ::=  $b = \text{esListaVacía (s)}$

## 1.3.2. Método constructivo (operacional).

### NOMBRE

Pila[T]

### CONJUNTOS

S Conjunto de pilas

T Conjunto de elementos

Bool Conjunto de valores booleanos {true, false}

### SINTAXIS

crearPila:  $\rightarrow$  S

tope: S  $\rightarrow$  T

pop: S  $\rightarrow$  S

push: T x S  $\rightarrow$  S

esVacíaPila: S  $\rightarrow$  Bool

## 1.3.2. Método constructivo (operacional).

### SEMÁNTICA

$\forall t \in T; \forall s, r \in S; b \in \text{Bool}$

1. pre-crearPila () ::= true
2. post-crearPila (s) ::= s = crearLista
3. pre-tope (s) ::= NOT esListaVacía (s)
4. post-tope (s; t) ::= t = primero (s)
5. pre-pop (s) ::= NOT esListaVacía (s)
6. post-pop (s; r) ::= r = cola (s)
7. pre-push (t, s) ::= true
8. post-push (t, s; r) ::= r = concatenar (formarLista (t), s)
9. pre-esVacíaPila (s) ::= true
10. post-esVacíaPila (s; b) ::= b = esListaVacía (s)

## 1.3.2. Método constructivo (operacional).

- **Ejecución de la especificación:** comprobar precondiciones y postcondiciones de todas las operaciones de la expresión.
- **Ejemplos:** Pila[Natural]
  - a) `tope (push (4, pop (push (2, crearPila) ) ) )`
  - b) `esVacíaPila (push (2, pop (crearPila) ) )`



## 1.3.2. Método constructivo (operacional).

- ¿Cómo cambiaría la especificación para un tipo de datos Cola[T]?

### NOMBRE

Cola[T]

### CONJUNTOS

C Conjunto de colas

T Conjunto de elementos

Bool Conjunto de valores booleanos {true, false}

### SINTAXIS

crearCola:  $\rightarrow$  C

frente: C  $\rightarrow$  T

inserta: T x C  $\rightarrow$  C

resto: C  $\rightarrow$  C

esVacíaCola: C  $\rightarrow$  Bool

## 1.3.2. Método constructivo (operacional).

### Conclusiones:

- La especificación constructiva está limitada por la necesidad de **modelos subyacentes**.
- **No confundir** especificación con implementación.
- Las especificaciones **algebraicas** son más potentes: usan el razonamiento inductivo.
- Pero las especificaciones **constructivas** son más fáciles de incluir en los programas (p.ej., mediante asertos).