# Abstracciones y especificaciones

## Algoritmos y estructuras de datos I

José Antonio Hernández López[1]

[1]Departamento de Informática y Sistemas
Universidad de Murcia

22 de septiembre de 2025

# Índice

# Abstracción

```c
#include <stdio.h>

int main() {
    int numbers[] = {4, 8, 15, 16, 23, 42};
    int length = sizeof(numbers) / sizeof(numbers[0]);

    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += numbers[i];
    }
    double average = (double) sum / length;
    printf("Average: %.2f\n", average);

    // más código [...]
    // tengo que hacer la media de nuevo -> copio y pego el bucle ...
    sum = 0;
    for (int i = 0; i < length; i++) {
        sum += numbers[i];
    }
    average = (double) sum / length;
    printf("Average again: %.2f\n", average);

    return 0;
}
```

# Abstracción

En algún lugar de mi proyecto defino:

```
double calculate_average(int arr[], int length) {
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += arr[i];
    }
    return (double) sum / length;
}
```

# Abstracción

En algún lugar de mi proyecto defino:

```
double calculate_average(int arr[], int length) {
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += arr[i];
    }
    return (double) sum / length;
}
```

De esta manera, me olvido de los detalles de la implementación (es decir, bucle + división) y tengo un función, la cual

- Sé que hace: calcula la media de un array (nombre `calculate_array`)
- Sé cómo se usa: recibe como entrada un array con su longitud y devuelve un real.

# Abstracción

En algún lugar de mi proyecto defino:

```
double calculate_average(int arr[], int length) {
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += arr[i];
    }
    return (double) sum / length;
}
```

De esta manera, me olvido de los detalles de la implementación (es decir, bucle + división) y tengo un función, la cual

- Sé que hace: calcula la media de un array (nombre `calculate_array`)
- Sé cómo se usa: recibe como entrada un array con su longitud y devuelve un real.

# Abstracción

```c
#include <stdio.h>
// importo calculate_average

int main() {
    int numbers[] = {4, 8, 15, 16, 23, 42};
    int length = sizeof(numbers) / sizeof(numbers[0]);

    printf("Average: %.2f\n", calculate_average(numbers, length));
    // más código [...]
    printf("Average again: %.2f\n", calculate_average(numbers, length));

    return 0;
}
```

# Abstracción

## Definición

La **abstracción** es el proceso de ocultar información no necesaria y exponer una visión simplificada.

- Simplifica sistemas complejos
- Reduce la carga cognitiva
- Aumenta la legibilidad, el mantenimiento y extensibilidad del código

# Abstracción

## Definición

La **abstracción** es el proceso de ocultar información no necesaria y exponer una visión simplificada.

- Simplifica sistemas complejos
- Reduce la carga cognitiva
- Aumenta la legibilidad, el mantenimiento y extensibilidad del código

En el ejemplo,

- lo relevante es `double calculate_average(int arr[], int length)`
- lo oculto es el bucle + división (la implementación).

# Ejemplos de abstracción

Analogías:

- **Subir el volumen de la tele**. Le doy al botón de subir el volumen. Sé que si le doy al botón se va a subir el volumen pero no me interesa el cómo funciona.
- **Conducir un coche**. Cuando conduces un coche solo te preocupas del volante, acelerador, freno, embrague y palanca de cambios. No te preocupas de cómo funciona el coche internamente.
- **Ordenar una pizza**. Cuando estás en el Mano a Mano y ordenas una pizza, no ves ni te importa cómo el chef amasa la masa ni qué marca de queso se utiliza.

# Tipos de abstracción

## Tipos de abstracción

Los tipos más importantes son:

- **Abstracciones de procedimiento**: oculta los pasos específicos de un algoritmo. Expone un procedimiento o función al que puedes llamar.
- **Abstracciones de datos**: oculta cómo los datos son guardados y mantenidos. Expone las propiedades y operaciones relevantes.
- **Abstracciones de iterador**: oculta los detalles de cómo recorres los elementos de una colección.

# Abstracciones de procedimiento

```c
double calculate_average(int arr[], int length) {
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += arr[i];
    }
    return (double) sum / length;
}
```

---

```c
#include <stdio.h>
// importo calculate_average

int main() {
    int numbers[] = {4, 8, 15, 16, 23, 42};
    int length = sizeof(numbers) / sizeof(numbers[0]);

    printf("Average: %.2f\n", calculate_average(numbers, length));
    // más código [...]
    printf("Average again: %.2f\n", calculate_average(numbers, length));

    return 0;
}
```

# Abstracciones de datos

La abstracción de datos se consigue básicamente con el uso los **tipos abstractos de datos** (TADs).

## Definición

Los TADs describen un conjunto de operaciones bien definidas que juntas proporcionan una herramienta útil para la resolución de problemas.

# Abstracciones de datos

La abstracción de datos se consigue básicamente con el uso los **tipos abstractos de datos** (TADs).

## Definición

Los TADs describen un conjunto de operaciones bien definidas que juntas proporcionan una herramienta útil para la resolución de problemas.

Los TADs ***suelen ser*** colecciones (aunque no tiene por qué). Ejemplos incluyen:

- conjunto: representan el conjunto matemático (no se permiten repetidos)
- bolsa: como un conjunto pero se permite repetidos
- **lista**: secuencia de elementos cuyo orden se mantiene
- pila: colección que mantiene un orden LIFO
- grafo: representan la construcción matemática de grafo con nodos y aristas
- ...

# Ejemplo: lista de enteros

En C, lo que se haría es escribir un `.h` con la interfaz y un `.c` con la implementación.

```
// fichero list.h
// cabeceras públicas
#ifndef LIST_H
#define LIST_H

typedef struct List List;
List* list_create();

void list_append(List* list, int value);
void list_remove(List* list, int value);
void list_print(const List* list);
void list_destroy(List* list);

#endif
```

# Ejemplo: lista de enteros

```c
// fichero list.c
// contiene la implementación que se oculta
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

typedef struct Node {
    int value;
    struct Node* next;
} Node;
struct List {
    Node* head;
};

List* list_create() {
    // implementación
}

void list_append(List* list, int value) {
    // implementación
}

// [...]
```

# Ejemplo: lista de enteros

De esta manera, me olvido (oculto) de la implementación y con lo que hay en el `.h`, puedo construir listas y usarlas para resolver problemas.

```
#include "list.h"

int main() {
    List* mylist = list_create();

    list_append(mylist, 10);
    list_append(mylist, 20);
    list_append(mylist, 30);

    list_print(mylist); // [10, 20, 30]
    list_remove(mylist, 20);
    list_print(mylist); // [10, 30]

    list_destroy(mylist);
    return 0;
}
```

# Ejemplo: lista de enteros

Es importante distinguir entre:

- **Tipo abstracto de datos**: el concepto de *lista* con la definición de sus operaciones.
- **Estructura de datos**: la forma en que se ha implementado. En este caso es una *linked list* en la que cada nodo tiene un entero y un puntero al siguiente nodo.

# Ejemplo: lista de enteros

En C++ se implementa usando clases (es un lenguaje orientado a objetos):

```
#ifndef LIST_H
#define LIST_H
// fichero List.h
class List {
// cabeceras públicas
public:
    List();
    ~List();

    void append(int value);
    void remove(int value);
    void print() const;

private:
    // Implementación que se oculta
    struct Node {
        int value;
        Node* next;
        Node(int v) : value(v), next(nullptr) {}
    };
    Node* head;
};
#endif
```

# Ejemplo: lista de enteros

```cpp
// fichero List.cpp con las implementaciones
#include "List.h"
#include <iostream>

List::List() : head(nullptr) {}
List::~List() {
    Node* curr = head;
    while (curr) {
        Node* temp = curr;
        curr = curr->next;
        delete temp;
    }
}

void List::append(int value) {
    // ...
}
void List::remove(int value) {
    // ...
}
void List::print() const {
    // ...
}
```

# Ejemplo: lista de enteros

```
#include "List.h"

int main() {
    List myList;

    myList.append(10);
    myList.append(20);
    myList.append(30);

    myList.print(); // [10, 20, 30]

    myList.remove(20);

    myList.print(); // [10, 30]

    return 0;
}
```

# Parametrización de tipo

La idea de tener un TAD para cada tipo es tedioso... En el caso de las listas:

- TAD lista de enteros
- TAD lista de carácteres
- TAD lista de reales
- ...

El concepto de lista es siempre el mismo y lo único que cambia es el tipo.

# Parametrización de tipo

La idea de tener un TAD para cada tipo es tedioso... En el caso de las listas:

- TAD lista de enteros
- TAD lista de carácteres
- TAD lista de reales
- ...

El concepto de lista es siempre el mismo y lo único que cambia es el tipo.

**Definición**

La **parametrización de tipos** en TADs es un mecanismo que permite definir un TAD de forma genérica, independiente del tipo concreto de los datos que maneja.

En el ejemplo, solo se definiría un TAD lista de $T$, donde $T$ es un tipo arbitrario.

# Parametrización de tipo

Hay lenguajes de programación que soportan este tipo de parametrización: **C++**, Java, C#, Go, ...

```
template <typename T>
class List {
public:
    List();
    ~List();

    void append(const T& value);
    void remove(const T& value);
    void print() const;

private:
    // Implementación oculta
    struct Node {
        T value;                    // ahora puede ser int, char, string, etc.
        Node* next;
        Node(const T& v) : value(v), next(nullptr) {}
    };
    Node* head;
};
#endif
```

# Parametrización de tipo

```
int main() {
    List<int> listaEnteros;
    listaEnteros.append(10);
    listaEnteros.append(20);
    listaEnteros.append(30);
    listaEnteros.print();   // 10 20 30

    listaEnteros.remove(20);
    listaEnteros.print();   // 10 30

    List<std::string> listaCadenas;
    listaCadenas.append("hola");
    listaCadenas.append("mundo");
    listaCadenas.print();   // hola mundo

    return 0;
}
```

# Abstracciones de datos

En los lenguajes de programación más usados, los tipos abstractos de datos más populares (colas, conjuntos, listas, etc.) ya nos lo dan implementados (con su parametrización de tipo). Si queremos usarlos nos basta con importarlos. Por ejemplo, en C++, en la Standard Template Library (STL) podemos encontrar:

| **TAD** | **Clase STL más cercana** | **Propiedades** |
|---|---|---|
| Bolsa (Bag) | `std::multiset` | Almacena elementos en orden ascendente, permite duplicados. |
| Conjunto (Set) | `std::set` | Almacena elementos únicos en orden ascendente. |
| Lista (List) | `std::list` | Lista doblemente enlazada, inserciones/eliminaciones en cualquier posición en tiempo constante. |
| Vector | `std::vector` | Arreglo dinámico con acceso aleatorio, eficiente para `push_back`. |
| Mapa (Map) | `std::map` | Pares clave–valor en orden ascendente, claves únicas. |
| Multimapa (Multimap) | `std::multimap` | Pares clave–valor en orden ascendente, permite claves duplicadas. |
| Conjunto no ordenado | `std::unordered_set` | Elementos únicos, sin orden, usa tablas hash. |
| Bolsa no ordenada | `std::unordered_multiset` | Permite duplicados, sin orden, usa tablas hash. |

Cuadro: Equivalencia entre conceptos y contenedores de la STL en C++, con enlaces a la documentación.

# Ejemplo: lista de enteros

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> numbers;

    numbers.push_back(3);
    numbers.push_back(1);
    numbers.push_back(3);
    numbers.push_back(2);

    std::cout << "List contents: ";
    for (int x : numbers) {
        std::cout << x << " ";
    }
    // [3, 1, 3, 2]
}
```

# Abstracciones de iterador

Los TADs más importantes suelen ser colecciones de elementos. Así pues, necesitamos mecanimos/abstracciones que hagan lo siguiente:

- para cada elemento de una colección, hacer algo usando ese elemento
- para cada elemento de una colección que cumpla una condición, hacer algo usando ese elemento

# Abstracciones de iterador

Los TADs más importantes suelen ser colecciones de elementos. Así pues, necesitamos mecanimos/abstracciones que hagan lo siguiente:

- para cada elemento de una colección, hacer algo usando ese elemento
- para cada elemento de una colección que cumpla una condición, hacer algo usando ese elemento

Para eso se usan las **abstracciones de iterador**. Hay dos tipos:

- Iterador como una abstracción funcional
- Iterador como una abstracción de datos

# Iterador como una abstracción funcional

En C++, los objetos instanciados con clases de la STL, se puede hacer:

```
#include <list>
#include <algorithm>
#include <iostream>

int main() {
    std::list<int> l{1,2,3,4};

    for (int x : l)
        std::cout << x << " ";
    std::cout << "\n";
}
```

Esto es una **abstracción funcional** ya que estás diciendo, para cada elemento del vector, haz algo sin preocuparte de qué está pasando internamente.

# Iterador como abstracción de datos

En C++, sobre los objetos instanciados con clases de la STL, se puede hacer:

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> l{1, 2, 3, 4};

    // Definimos un iterador para la lista
    std::list<int>::iterator it;

    // Recorremos la lista usando el iterador
    for (it = l.begin(); it != l.end(); ++it) {
        std::cout << *it << " "; // *it accede al valor
    }

    std::cout << "\n";
    return 0;
}
```

Este caso es iterador como **abstracción de datos** ya que creamos un objeto de tipo iterador y lo usamos para iterar sobre el vector.

# Iterador como abstracción de datos

Equivalente a lo anterior pero con un `while`:

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> l{1, 2, 3, 4};

    // Definimos un iterador para la lista
    std::list<int>::iterator it = l.begin();

    // Recorremos la lista usando el iterador con while
    while (it != l.end()) {
        std::cout << *it << " "; // *it accede al valor
        ++it;
    }

    std::cout << "\n";
    return 0;
}
```

# Iterador como abstracción de datos

Los iteradores como abstracción de datos son útiles para determinar el orden del recorrido. Por ejemplo, podemos recorrer la lista al revés.

```
#include <iostream>
#include <list>

int main() {
    std::list<int> l{1, 2, 3, 4};

    // Definimos un iterador inverso para la lista
    std::list<int>::reverse_iterator rit;

    // Recorremos la lista en orden inverso usando el iterador inverso
    for (rit = l.rbegin(); rit != l.rend(); ++rit) {
        std::cout << *rit << " "; // *rit accede al valor
    }

    std::cout << "\n";
    return 0;
}
```

# Especificaciones

## Definición

Las **especificaciones** son la descripción de qué hace un módulo, función, clase o tipo abstracto de datos sin entrar en cómo se implementa.

Hay dos tipos:

- Informales: escritas en lenguaje natural
- Formales: escritas en lenguaje matemático

# Especificaciones informales

**Definición**

Una especificación informal es una descripción escrita en lenguaje natural (como el español o el inglés) que explica qué debe hacer un programa, módulo o función, sin utilizar un lenguaje matemático riguroso ni una sintaxis estricta.

- Fáciles de leer y entender por personas no expertas en formalismos.
- Ambiguas o incompletas en ocasiones, ya que el lenguaje natural puede dar lugar a diferentes interpretaciones.

# Especificaciones informales en abstracciones funcionales

## Sintaxis especificaciones informales en abstracciones funcionales

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

# Especificaciones informales en abstracciones funcionales

**Ejemplo 1**. Eliminar la repetición en los elementos de un array
**Operación** quitarDuplicados (**ent** *a*: array[entero])

- Modifica: *a*
- Calcula: Quita los elementos repetidos de *a*. Puede ocurrir que se modifique el tamaño de *a* (haciéndose más pequeño).

---

# Especificaciones informales en abstracciones funcionales

**Ejemplo 1**. Eliminar la repetición en los elementos de un array
**Operación** quitarDuplicados (**ent** *a*: array[entero])

- Modifica: *a*
- Calcula: Quita los elementos repetidos de *a*. Puede ocurrir que se modifique el tamaño de *a* (haciéndose más pequeño).

---

**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 $i = -1$.

# Especificaciones informales en abstracciones funcionales

Hay veces que se tiene que generalizar la especificación para todo tipo $T$.

# Especificaciones informales en abstracciones funcionales

Hay veces que se tiene que generalizar la especificación para todo tipo $T$.
**Ejemplo 1**. Eliminar la repetición en los elementos de un array.
**Operación** quitarDuplicados [$T$: tipo](**ent** $a$: array[$T$])

- Modifica: $a$
- Calcula: Quita los elementos repetidos de $a$. Puede ocurrir que se modifique el tamaño de $a$ (haciéndose más pequeño).

---

# Especificaciones informales en abstracciones funcionales

Hay veces que se tiene que generalizar la especificación para todo tipo $T$.
**Ejemplo 1**. Eliminar la repetición en los elementos de un array.
**Operación** quitarDuplicados [$T$: tipo](**ent** $a$: array[$T$])

- Modifica: $a$
- Calcula: Quita los elementos repetidos de $a$. Puede ocurrir que se modifique el tamaño de $a$ (haciéndose más pequeño).

---

**Ejemplo 2**. Buscar un elemento en un array.
**Operación** buscar [$T$: tipo] (**ent** $a$: array[$T$]; $x$: $T$; **sal** $i$: entero)

- Requiere: $T$ de ser un tipo con una relación de orden definida. $a$ 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 $i = -1$.

# Especificaciones informales en abstracciones funcionales

Ejemplos reales:

- `push_back` de las listas en C++: `https://en.cppreference.com/w/cpp/container/list/push_back.html`
- `np.mean` en numpy de Python `https://numpy.org/devdocs/reference/generated/numpy.mean.html`

# Especificaciones informales en abstracciones de datos

## Sintaxis especificaciones informales en TADs

**TAD** <nombre> es <lista_operaciones>
**Descripción:**
Descripción textual del tipo
**Operaciones:**
Especificación informal de las operaciones de la lista anterior siguiendo la sintaxis de las especificaciones de abstracciones funcionales.

# Especificaciones informales en 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[$T$])
  - **Calcula**: Devuelve un conjunto de tipo $T$ vacío.
- **Operación** Insertar (**ent** $c$: Conjunto[$T$]; $x$: $T$)
  - **Modifica**: $c$.
  - **Calcula**: añade el elemento $x$ al conjunto $c$.
- ...

# Especificaciones informales en abstracciones de iteradores

En iterador como abstracción funcional:

# Especificaciones informales en abstracciones de iteradores

En iterador como 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 $x$ del conjunto $c$, aplicando sobre ellos la operación acción($x$).

```
int main() {
    std::set<int> s{1,2,3,4};

    for (int x : s)
        std::cout << x << " ";
}
```

# Especificaciones informales en abstracciones de iteradores

En iterador como TAD:

# Especificaciones informales en abstracciones de iteradores

En iterador como TAD:

**TAD** IteradorReverso[$T$: tipo] es Iniciar, Actual, Avanzar, EsFinal
**Descripción**
Los valores de tipo IteradorReverso[$T$] son iteradores definidos sobre listas de cualquier tipo. El iterador se debe inicializar con Iniciar.
**Operaciones**

- **Operación** Iniciar (**ent** $l$: List[$T$]; **sal** it: IteradorReverso)
  - **Calcula**: devuelve un iterador nuevo colocándose al final de la lista $l$.
- **Operación** Actual (**ent** iter: IteradorReverso; **sal** t: $T$)
  - **Calcula**: devuelve el elemento al que apunta el iterador actualmente.
- ...

# Especificaciones informales en abstracciones de iteradores

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> l{1, 2, 3, 4};

    // Iniciar
    std::list<int>::reverse_iterator rit = l.rbegin();

    // mientras no EsFinal
    while (rit != l.rend()) {
        // imprimimos el valor actual, *it -> Actual
        std::cout << *rit << " "; // *rit accede al valor
        //Avanzar
        ++rit;
    }

    std::cout << "\n";
    return 0;
}
```

# Especificaciones formales

**Definición**

Una especificación formal es una descripción precisa y matemática del comportamiento que debe tener un sistema software o un componente (por ejemplo, un tipo abstracto de datos o un módulo).

- Precisión y ausencia de ambigüedad.
- Basada en lógica, teoría de conjuntos, álgebra, etc.
- Permite razonamiento matemático: verificación, pruebas de corrección, demostración de propiedades.
- Independiente de la implementación.

# Especificaciones formales

Tipos de especificaciones:

- **Método axiomático o algebraico**: describir un sistema a través de axiomas y ecuaciones que definen las relaciones entre operaciones.
- Método constructivo: para cada operación, se establecen las precondiciones y las postcondiciones.

# Método axiomático

Se suele aplicar sobre TADs. La descripción formal constará de:

- Nombre del TAD
- Conjuntos matemáticos involucrados
- Sintaxis de las operaciones
- Semántica de las operaciones como conjunto de axiomas

# Método axiomático

**Nombre**: Natural
**Conjuntos involucrados**: $\mathbf{N}$, $\mathbf{B} := \{\texttt{true}, \texttt{false}\}$
**Sintaxis**:

- cero: $\rightarrow \mathbf{N}$
- sucesor: $\mathbf{N} \rightarrow \mathbf{N}$
- suma: $\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$
- esCero: $\mathbf{N} \rightarrow \mathbf{B}$
- esIgual: $\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$

# Método axiomático

Semántica: $\forall n, m \in \mathbf{N}$

- $\text{suma}(\text{cero}, n) = n$
- $\text{suma}(\text{sucesor}(m), n) = \text{sucesor}(\text{suma}(n, m))$
- $\text{esCero}(\text{cero}) = \text{true}$
- $\text{esCero}(\text{sucesor}(n)) = \text{false}$
- $\text{eslgual}(\text{cero}, n) = \text{esCero}(n)$
- $\text{eslgual}(\text{sucesor}(n), \text{cero}) = \text{false}$
- $\text{eslgual}(\text{sucesor}(n), \text{sucesor}(m)) = \text{eslgual}(n, m)$

# Método axiomático

Tipos de operaciones:

- Constructores: son las operaciones que generan todos los elementos posibles del TAD.
  Ejemplo: cero y sucesor.

# Método axiomático

Tipos de operaciones:

- Constructores: son las operaciones que generan todos los elementos posibles del TAD.
  Ejemplo: cero y sucesor.
- Modificación: a partir de un valor del tipo, obtienen otro valor del tipo TAD, y no son constructores.
  Ejemplo: suma.

# Método axiomático

Tipos de operaciones:

- Constructores: son las operaciones que generan todos los elementos posibles del TAD.
  Ejemplo: cero y sucesor.
- Modificación: a partir de un valor del tipo, obtienen otro valor del tipo TAD, y no son constructores.
  Ejemplo: suma.
- Consulta: Devuelven un valor que no es del tipo TAD.
  Ejemplo: esCero y esIgual.

# Método axiomático

## Ejecución de una especificación algebraica

Aplicar sucesivamente las reglas de la semántica hasta alcanzar una forma normal (una forma a la que no se le pueden aplicar más reglas).

- $(1 + 0) + 1$, suma(suma(sucesor(cero),cero), sucesor (cero))
- $2 = (1 + 0) + 1$ ?, eslgual (sucesor (sucesor (cero)), suma (suma (sucesor (cero), cero), sucesor (cero) ) )

# Método axiomático

Los axiomas de la semántica deben satisfacer:

- Completitud: Una especificación es incompleta, si hay operaciones cuyo resultado no esté del todo determinado.

# Método axiomático

Los axiomas de la semántica deben satisfacer:

- Completitud: Una especificación es incompleta, si hay operaciones cuyo resultado no esté del todo determinado.
  Ejemplo: esto es incompleto:
  - $\text{esPar}(\text{cero}) = \text{true}$
  - $\text{esPar}(\text{sucesor}(\text{sucesor}(n))) = \text{esPar}(n)$

  Intenta calcular $\text{esPar}(\text{sucesor}(\text{cero}))$.

# Método axiomático

Los axiomas de la semántica deben satisfacer:

- Completitud: Una especificación es incompleta, si hay operaciones cuyo resultado no esté del todo determinado.
  Ejemplo: esto es incompleto:
  - $\text{esPar}(\text{cero}) = \text{true}$
  - $\text{esPar}(\text{sucesor}(\text{sucesor}(n))) = \text{esPar}(n)$

  Intenta calcular $\text{esPar}(\text{sucesor}(\text{cero}))$.
- Consistencia: Una especificación es consistente si no se pueden derivar contradicciones a partir de los axiomas.

# Método axiomático

Los axiomas de la semántica deben satisfacer:

- Completitud: Una especificación es incompleta, si hay operaciones cuyo resultado no esté del todo determinado.
  Ejemplo: esto es incompleto:
  - esPar(cero) = true
  - esPar(sucesor(sucesor($n$))) = esPar($n$)

  Intenta calcular esPar(sucesor(cero)).
- Consistencia: Una especificación es consistente si no se pueden derivar contradicciones a partir de los axiomas.
  Ejemplo: si añadiera la regla esCero(cero) = false, obtendría un sistema inconsistente.

# Método axiomático

## TAD Pila

El TAD Pila representa una colección de elementos con un orden de acceso LIFO (Last In, First Out), es decir, el último elemento en entrar es el primero en salir.

# Método axiomático

## TAD Pila

El TAD Pila representa una colección de elementos con un orden de acceso LIFO (Last In, First Out), es decir, el último elemento en entrar es el primero en salir.

**Nombre**: Pila$[T]$

# Método axiomático

## TAD Pila

El TAD Pila representa una colección de elementos con un orden de acceso LIFO (Last In, First Out), es decir, el último elemento en entrar es el primero en salir.

**Nombre**: Pila$[T]$
**Conjuntos involucrados**:

- $\mathbf{P}$: conjunto de pilas
- $\mathbf{B} := \{\texttt{true}, \texttt{false}\}$: booleanos
- $T$: conjunto de elementos que se pueden meter en la pila
- $\mathbf{M} := \{\text{Error: pila vacía}\}$: conjunto de mensajes de error

# Método axiomático

**Sintaxis**:

- Constructores:
  - pilaVacia: $\rightarrow \mathbf{P}$
  - push: $T \times \mathbf{P} \rightarrow \mathbf{P}$
- Modificadores:
  - pop: $\mathbf{P} \rightarrow \mathbf{P}$
- Consulta:
  - esVacía: $\mathbf{P} \rightarrow \mathbf{B}$
  - tope: $\mathbf{P} \rightarrow T \cup \mathbf{M}$

# Método axiomático

| | **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)) = ? |

# Método axiomático

**Semántica:**

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

1. $\text{esVacía}(\text{pilaVacía}) = \text{true}$
2. $\text{esVacía}(\text{push}(t, p)) = \text{false}$
3. $\text{pop}(\text{pilaVacía}) = \text{pilaVacía}$
4. $\text{pop}(\text{push}(t, p)) = p$
5. $\text{tope}(\text{pilaVacía}) = \text{Error: pila vacía}$
6. $\text{tope}(\text{push}(t, p)) = t$

# Método axiomático

Ejercicios:

1. $\text{pop}(\text{push}(3, \text{push}(2, \text{pop}(\text{pilaVacía}))))$
2. $\text{tope}(\text{pop}(\text{push}(1, \text{push}(2, \text{pilaVacía}))))$
3. Añadir una operación **esIgual**
4. Modificar la sintaxis y semántica para que **pop** devuelva el primer elemento y lo saque de la pila

# Método axiomático

**Nombre**: Bolsa$[T]$

**Conjuntos involucrados**:

- $B$: conjunto de bolsas
- $\mathbf{B} := \{\texttt{true}, \texttt{false}\}$: booleanos
- $T$: conjunto de elementos que se pueden meter en la bolsa
- $\mathbf{N}$: conjunto de naturales previamente definidos

**Sintaxis**:

- Constructores:
  - bolsaVacía: $\rightarrow B$
  - poner: $T \times B \rightarrow B$
- Consulta:
  - esVacía: $B \rightarrow \mathbf{B}$
  - tamaño: $B \rightarrow \mathbf{N}$
  - cuantos: $T \times B \rightarrow \mathbf{N}$ (se puede usar if-else)

# Método axiomático

**Nombre**: Bolsa$[T]$

**Conjuntos involucrados**:

- $B$: conjunto de bolsas
- $\mathbf{B} := \{\texttt{true, false}\}$: booleanos
- $T$: conjunto de elementos que se pueden meter en la bolsa
- $\mathbf{N}$: conjunto de naturales previamente definidos

**Sintaxis**:

- Constructores:
  - bolsaVacía: $\rightarrow B$
  - poner: $T \times B \rightarrow B$
- Consulta:
  - esVacía: $B \rightarrow \mathbf{B}$
  - tamaño: $B \rightarrow \mathbf{N}$
  - cuantos: $T \times B \rightarrow \mathbf{N}$ (se puede usar if-else)

Vamos también a especificar: quitar, quitarTodos, estáEn y esIgual.

# Método axiomático

El método axiomático (y el constructivo) se aplica en software crítico, donde un fallo puede costar vidas o millones porque proporciona precisión, verificabilidad y confianza matemática:

- Aviónica
- Sistemas ferroviarios
- Industria nuclear

# Método axiomático

El método axiomático (y el constructivo) se aplica en software crítico, donde un fallo puede costar vidas o millones porque proporciona precisión, verificabilidad y confianza matemática:

- Aviónica
- Sistemas ferroviarios
- Industria nuclear

En la práctica, para diseñar estos sistemas, se hace lo siguiente:

1. Extraer requitos informales
2. Extraer modelo formal y verificar la consistencia ←
3. Implementación
4. Verificación de la implementación contra la especificación ←
5. Validación y pruebas empíricas
6. Certificación y mantenimiento

## Maude

Maude es un lenguaje (interpretado escrito en C++) de especificación formal y programación declarativa basado en la lógica de reescritura.
La reescritura significa transformar un término (una expresión) aplicando reglas que indican cómo sustituir una parte por otra.

Instalación:

```
>> wget http://maude.cs.uiuc.edu/maude1/current/system/maude-linux.tar.Z
>> gunzip -c maude-linux.tar.Z | tar -xvf -
>> cd maude-linux/bin
>> ./maude.linux
```

# Maude

**Nombre**: Natural
**Conjuntos involucrados**: $\mathbf{N}$, $\mathbf{B}$
**Sintaxis**:

- cero: $\rightarrow \mathbf{N}$
- sucesor: $\mathbf{N} \rightarrow \mathbf{N}$
- suma: $\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$
- esCero: $\mathbf{N} \rightarrow \mathbf{B}$
- eslgual: $\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$

**Semántica**:

...

# Maude

**Nombre**: Natural
**Conjuntos involucrados**: $\mathbf{N}$, $\mathbf{B}$
**Sintaxis**:

- cero: $\rightarrow \mathbf{N}$
- sucesor: $\mathbf{N} \rightarrow \mathbf{N}$
- suma: $\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$
- esCero: $\mathbf{N} \rightarrow \mathbf{B}$
- eslgual: $\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$

**Semántica**:

...

---

```
fmod NATURAL is

   protecting BOOL .
   sort N .

   op cero :  -> N .
   op sucesor : N -> N .
   op suma : N N -> N .
   op esCero : N -> Bool .
   op esIgual : N N -> Bool .
...
endfm
```

# Maude

**Semántica**:
$\forall n, m \in \mathbf{N}$

- $\text{esCero}(\text{cero}) = \text{true}$
- $\text{esCero}(\text{sucesor}(n)) = \text{false}$
- $\text{eslgual}(\text{cero}, n) = \text{esCero}(n)$
- $\text{eslgual}(\text{sucesor}(n), \text{cero}) = \text{false}$
- $\text{eslgual}(\text{sucesor}(n), \text{sucesor}(m)) = \text{eslgual}(n, m)$
- $\text{suma}(\text{cero}, n) = n$
- $\text{suma}(\text{sucesor}(m), n) = \text{sucesor}(\text{suma}(n, m))$

# Maude

**Semántica**:
$\forall n, m \in \mathbf{N}$

- esCero(cero) = true
- esCero(sucesor($n$)) = false
- esIgual(cero, $n$) = esCero($n$)
- esIgual(sucesor($n$), cero) = false
- esIgual(sucesor($n$), sucesor($m$)) = esIgual($n$, $m$)
- suma(cero, $n$) = $n$
- suma(sucesor($m$), $n$) = sucesor(suma($n$, $m$))

---

```
fmod NATURAL is
    ...
    vars n m : N .
    eq esCero (cero) = true .
    eq esCero (sucesor (n)) = false .
    eq esIgual(cero, n) = esCero(n) .
    eq esIgual(sucesor(n), cero) = false .
    eq esIgual(sucesor(n), sucesor(m)) = esIgual (n, m) .
    eq suma (cero, n) = n .
    eq suma (sucesor (m), n) = sucesor (suma (m, n)) .
endfm
```

# Maude

- `fmod` significa módulo en Maude y es lo equivalente a TAD
- `sort` se usa para definir un tipo de dato dentro de un módulo (por ejemplo, `sort N .`)
- `protecting` se usa para importar un `fmod` definido anteriormente (por nosotros o en la librería de *utilidades* de Maude)
- `op` (operator) se usa para definir la sintaxis de las operaciones
- `eq` (equation) se usa para definir los axiomas
- `vars` (variables) se usa para definir las variables usadas en los axiomas (equivalente al $\forall$)
- La extensión de Maude es `.maude`

# Maude

Pipeline que se sigue:

1. Se define un `.maude` (e.g., `natural.maude`) con uno o más `fmod`
2. Se abre el intérprete (`./maude-linux/bin/maude.linux`)
3. Se carga el fichero en el intérprete (usando `in natural.maude`)
4. Usando `red` (reducción) y una expresión, se aplican los axiomas para reducir dicha a expresión a una forma normal.
   - `red esIgual(sucesor(cero), cero) .`
   - `red suma(suma(sucesor(cero), cero), sucesor(cero)) .`

# Maude

Consideraciones:

- Sintaxis muy estricta.
- Espacios en blanco necesarios: antes y después de `:`, de `->` y de `=`.
- Acabar las sentencias con `" ."` (espacio en blanco + punto).
- Comprobar los paréntesis.
- Si hay un fallo puede pasar de todo: lo indica bien, da un fallo en un sitio extraño, se queda colgado, etc.
- Usad VSCode con algún plugin de Maude.

# Método constructivo

En este método, se definen de manera matemática/formal:

- Precondiciones: condiciones que debe satisfacer la entrada para que se dé el comportamiento deseado.
- Postcondiciones: relaciones que se cumplen cuando se ejecuta la operación.

# Método constructivo

En este método, se definen de manera matemática/formal:

- Precondiciones: condiciones que debe satisfacer la entrada para que se dé el comportamiento deseado.
- Postcondiciones: relaciones que se cumplen cuando se ejecuta la operación.

Se asumen dos roles:

- Implementador: asumir que se cumplen las precondiciones y asegurar que se cumple la postcondición.
- Usuario: asegurar el cumplimiento de las precondiciones y asumir que se cumple la postcondición.

# Método constructivo

**Ejemplo**. Operación **máximo**, que tiene como entrada dos números reales **positivos** y da como salida el mayor de los dos.
**Sintaxis**:
máximo: $\mathbf{R} \times \mathbf{R} \to \mathbf{R}$
**Semántica**:

- **pre-máximo**$(x, y) ::= x \geq 0$ y $y \geq 0$
- **post-máximo**$(x, y; r) ::= r \geq x$ y $r \geq y$ y $(x = r$ o $y = r)$

# Método constructivo

**Ejemplo**. Operación **máximo**, que tiene como entrada dos números reales **positivos** y da como salida el mayor de los dos.
**Sintaxis**:
máximo: $\mathbf{R} \times \mathbf{R} \to \mathbf{R}$
**Semántica**:

- **pre-máximo**$(x, y) ::= x \geq 0$ y $y \geq 0$
- **post-máximo**$(x, y; r) ::= r \geq x$ y $r \geq y$ y $(x = r$ o $y = r)$

La idea es:

- El implementador asume que son números positivos y asegura que la poscondición se cumple.
- El usuario, si quiere usar la función y obtener el resultado esperado, debe asegurarse de usarla con números positivos.

# Método constructivo

Este tipo de especificaciones se pueden implementar fácilmente en C y C++ mediante el uso de asertos.

## Definición

Los asertos son comprobaciones que el programador introduce en el código para verificar que ciertas condiciones siempre se cumplen en tiempo de ejecución.

```
#include <iostream>
#include <cassert>  // Para assert

double maximop(double x, double y) {
    assert(x >= 0 && y >= 0); // Precondición: ambos números no negativos
    double r;
    if (x > y)
        r = x;
    else
        r = y;
    // Postcondición: r es mayor o igual que ambos, y coincide con uno de ellos
    assert(r >= x && r >= y && (r == x || r == y));
    return r;
}
```

# Método constructivo

- Cuando definimos un TAD sencillo (por ejemplo, números reales o naturales), podemos dar pre y pos-condiciones de manera sencilla.
- Pero en un TAD más complejo, necesitamos un **modelo subyacente**.
- **Ejemplo**: ¿cómo definimos TAD Pila[T] por el método constructivo?

# Método constructivo

- Cuando definimos un TAD sencillo (por ejemplo, números reales o naturales), podemos dar pre y pos-condiciones de manera sencilla.
- Pero en un TAD más complejo, necesitamos un **modelo subyacente**.
- **Ejemplo**: ¿cómo definimos TAD Pila[T] por el método constructivo?

## Idea

Definimos el TAD Lista[T] por el método axiomático y lo usamos como modelo subyacente para definir Pila[T]. **En esencia, una pila se puede ver como una lista con la restricción LIFO.**

# Método constructivo

**Nombre:** Lista[T]

**Conjuntos**

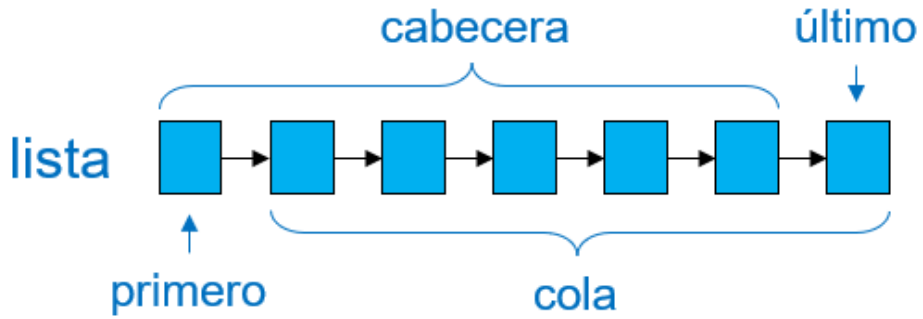
- $\mathbf{L}$: Conjunto de listas
- $T$: Conjunto de elementos
- $\mathbf{B}$: Conjunto de booleanos $\{true, false\}$
- $\mathbf{N}$: Conjunto de naturales
- $\mathbf{M}$: Conjunto de mensajes {"La lista está vacía"}

# Método constructivo

**Sintaxis:**

$$
\begin{array}{ll}
\text{crearLista:} & \rightarrow \mathbf{L} \\
\text{formarLista:} & T \rightarrow \mathbf{L} \\
\text{concatenar:} & \mathbf{L} \times \mathbf{L} \rightarrow \mathbf{L}
\end{array}
$$

$$
\begin{array}{ll}
\text{último:} & \mathbf{L} \rightarrow T \cup \mathbf{M} \\
\text{cabecera:} & \mathbf{L} \rightarrow \mathbf{L} \\
\text{primero:} & \mathbf{L} \rightarrow T \cup \mathbf{M} \\
\text{cola:} & \mathbf{L} \rightarrow \mathbf{L} \\
\text{longitud:} & \mathbf{L} \rightarrow \mathbf{N} \\
\text{esListaVacía:} & \mathbf{L} \rightarrow \mathbf{B}
\end{array}
$$



# Método constructivo

**Semántica:**

$$\forall t \in T;\ \forall a, b \in \mathbf{L}$$

1. último(crearLista) = "La lista está vacía"
2. último(formarLista(t)) = t
3. último(concatenar(a,b)) = $\begin{cases} \text{último}(a) & \text{si esListaVacía}(b) \\ \text{último}(b) & \text{en otro caso} \end{cases}$
4. cabecera(crearLista) = crearLista
5. cabecera(formarLista(t)) = crearLista

# Método constructivo

6. cabecera(concatenar(a,b)) =
$$\begin{cases} \text{cabecera}(a) & \text{si esListaVacía}(b) \\ \text{concatenar}(a, \text{cabecera}(b)) & \text{en otro caso} \end{cases}$$
7. primero(crearLista) = "La lista está vacía"
8. primero(formarLista(t)) = t
9. primero(concatenar(a,b)) =
$$\begin{cases} \text{primero}(b) & \text{si esListaVacía}(a) \\ \text{primero}(a) & \text{en otro caso} \end{cases}$$

# Método constructivo

10. cola(crearLista) = crearLista
11. cola(formarLista(t)) = crearLista
12. cola(concatenar(a,b)) = $\begin{cases} \text{cola}(b) & \text{si esListaVacía}(a) \\ \text{concatenar}(\text{cola}(a), b) & \text{en otro caso} \end{cases}$
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) $\wedge$ esListaVacía(b)

# Método constructivo

**Nombre**: Pila$[T]$

**Conjuntos involucrados**: $\mathbf{P}$, $\mathbf{B} := \{\texttt{true}, \texttt{false}\}$: , $T$, ~~$\mathbf{M} := \{\text{Error: pila vacía}\}$~~

**Sintaxis**:

- pilaVacia: $\rightarrow \mathbf{P}$
- push: $T \times \mathbf{P} \rightarrow \mathbf{P}$
- pop: $\mathbf{P} \rightarrow \mathbf{P}$
- esVacía: $\mathbf{P} \rightarrow \mathbf{B}$
- tope: $\mathbf{P} \rightarrow T \cup \mathbf{M}$

# Método constructivo

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

1. pre-pilaVacia() ::= true
2. post-pilaVacia($s$) ::= $s$ = crearLista
3. pre-tope($s$) ::= $\neg$ esListaVacía($s$)
4. post-tope($s; t$) ::= $t$ = primero($s$)
5. pre-pop($s$) ::= $\neg$ 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-esVací($s$) ::= true
10. post-esVacía($s; b$) ::= $b$ = esListaVacía($s$)

# Método constructivo

Ejecución de una especificación constructiva:

**Ejecución de una especificación constructiva**

Comprobar precondiciones y postcondiciones de todas las operaciones de la expresión.

Ejemplos:

1. $\text{tope}(\text{push}(4,\ \text{pop}(\ \text{push}(2,\ \text{pilaVacia}))))$
2. $\text{esVacía}(\text{push}(2,\ \text{pop}(\ \text{pilaVacia})))$

# Conclusiones

- Abstracción: proceso de ocultar información innecesaria y exponer una visión simplificada.
- Tres tipos de abstracción: procedimiento/funcional, de datos (tipos abstractos de datos, TADs) e iterador.
- Especificación: descripción de qué hace un módulo, función, clase, tipo abstracto de datos sin entrar en cómo se implementa.
- Dos tipos de especificaciones: informales y formales.
- Las más usadas en la práctica son las informales.
- Las formales se suelen usan en software crítico porque proporcionan precisión, verificabilidad y confianza matemática en sistemas donde los errores son inaceptables.
- Tipos de especificaciones formales: axiomáticas y constructivas.