Capítulo 0
Problemas, Programas, Estructuras y Algoritmos

Contenido del Capítulo:

0.1. RESOLUCIÓN DE PROBLEMAS

La habilidad de programar ordenadores constituye una de las bases necesarias de todo informático. Pero la característica fundamental de un informático es su capacidad para **resolver problemas**. Eso implica comprender y analizar las necesidades de los problemas, saber modelarlos de forma abstracta diferenciando lo importante de lo irrelevante, disponer de una variada batería de herramientas conceptuales que se puedan aplicar en su resolución, trasladar un diseño abstracto a un lenguaje y entorno concretos y, finalmente, ser capaz de evaluar la corrección, prestaciones y posibles inconvenientes de la solución planteada.

Herramientas del desarrollo de programas: ***estructuras de datos y algoritmos***. Las estructuras de datos representan la parte estática de la solución al problema, el componente almacenado, donde se encuentran los datos de entrada, de salida y los necesarios para posibles cálculos intermedios. Los algoritmos representan la parte dinámica del sistema, el componente que manipula los datos para obtener la solución. Algoritmos y estructuras de datos se relacionan de forma muy estrecha: las estructuras de datos son manipuladas mediante algoritmos que añaden o modifican valores en las mismas; y cualquier algoritmo necesita manejar datos que estarán almacenados en cierta estructura. La idea se puede resumir en la fórmula magistral de Niklaus Wirth: Algoritmos + Estructuras de datos = Programas.

En ciertos ámbitos de aplicación predominará la componente algorítmica –como, por ejemplo, en problemas de optimización y cálculo numérico– y en otros la componente de estructuras –como en entornos de bases de datos y sistemas de información–, pero cualquier aplicación requerirá siempre de ambos. Esta dualidad aparece en el nivel abstracto: un tipo abstracto está formado por un dominio de valores (estructura abstracta) y un conjunto de operaciones (algoritmos abstractos). Y la dualidad se refleja también en el nivel de implementación: un módulo (en lenguajes estructurados) o una clase (en lenguajes orientados a objetos) están compuestos por una estructura de atributos o variables, y una serie de procedimientos de manipulación de los anteriores.

PASOS QUE CONSTITUYEN EL PROCESO DE RESOLUCIÓN DE PROBLEMAS.

El proceso de resolución de problemas parte siempre de un **problema**, de un enunciado más o menos claro que alguien plantea porque le vendría bien que estuviera resuelto.

<u>Primer paso</u> en el proceso de resolución de problemas, en programación:

- ***análisis de las características del problema (comprensión del problema)*.** El análisis debe producir como resultado un **modelo abstracto** del problema. El modelo abstracto es un modelo conceptual, una abstracción del problema, que reside exclusivamente en la mente del individuo y donde se desechan todas las cuestiones irrelevantes para la resolución del problema.

En ese **modelo abstracto** nos quedamos con lo esencial del problema. Normalmente, este modelo se crea a través de una **analogía** con algo conocido previamente. Por ejemplo, para enseñar a un alumno inexperto lo que es un algoritmo (concepto para él desconocido) se utiliza la analogía con una receta de cocina (concepto conocido, esperemos), y las estructuras de datos se asimilan con la disposición de armarios, cajones y recipientes donde se guardan los ingredientes y el bizcocho resultante.

<u>Segundo paso</u> en el proceso de resolución de problemas, en programación:

- ***diseño de una solución***

Una vez que tengamos un modelo completo y adecuado, significará que hemos entendido bien el problema. Como hemos visto en los ejemplos, el modelo conlleva también un algoritmo informal. El siguiente paso de refinamiento consistiría en diseñar una solución. El **diseño de un programa** es equivalente a los planos de un arquitecto: el diseño describe el aspecto que tendrá el programa, los materiales que se necesitan y dónde y cómo colocarlos. El diseño es siempre un paso previo a la implementación, pero que se olvida muy a menudo por los programadores primerizos.

Ejes básicos en los que se articula el desarrollo de un programa: estructuras de datos y algoritmos.

Estructuras de datos: determinan la forma de almacenar la información necesaria para resolver el problema.

Algoritmos: manipulan esa información para producir unos datos de salida a partir de unos datos de entrada. El objetivo último de la algorítmica es encontrar la mejor forma de resolver los problemas.

Los tipos de datos usados a nivel de diseño son **tipos abstractos**, en los cuales lo importante son las operaciones que ofrecen y no la representación en memoria. Los algoritmos son una versión refinada de los algoritmos abstractos del paso anterior. No obstante, son aún **algoritmos en pseudocódigo**, donde se dan cosas por supuestas. Definir los tipos de datos para almacenar la solución suele ser mucho más difícil que definirlos para los datos de entrada. Además, modificaciones en los algoritmos pueden requerir cambios en los tipos de datos y viceversa.

En resumen, el diseño indica qué cosas se deben programar, cómo se estructura la solución del problema, dónde colocar cada funcionalidad y qué se espera en concreto de cada parte en la que se ha descompuesto la solución. Esta descripción de lo que debe hacer cada parte es lo que se conoce como la **especificación**.

<u>Tercer paso</u> en el proceso de resolución de problemas, en programación:

- ***implementación del diseño***

La implementación parte del diseño previo. Una implementación será correcta si cumple la especificación dada en el paso anterior.

La dualidad tipos/algoritmos del diseño se traslada en la implementación a estructuras/algoritmos. Para cada uno de los tipos de datos del diseño, se debe elegir una estructura de datos adecuada. La elección debe seguir los criterios de eficiencia –en cuanto a tiempo y a uso de memoria– que se hayan especificado para el problema. En cada caso, la estructura más adecuada podrá ser una u otra.

Por otro lado, **la implementación de los algoritmos** también se puede ver como una serie de tomas de decisiones, para trasladar un algoritmo en pseudocódigo a un lenguaje concreto. Por el ejemplo, si se necesita repetir un cálculo varias veces, se puede usar un bucle for, un while, se puede usar recursividad e incluso se puede poner varias veces el mismo código si el número de ejecuciones es fijo.

Realmente, la implementación de estructuras de datos y algoritmos no van por separado. Ambas son simultáneas. De hecho, así ocurre con los mecanismos de los lenguajes de programación –módulos o clases– en los que se asocia cada estructura con los algoritmos que la manejan.

<u>Cuarto paso</u> en el proceso de resolución de problemas, en programación:

- ***verificación y evaluación del programa resultante***

La resolución de un problema no acaba con la implementación de un programa. Los programas deberían ser comprobados de forma exhaustiva, verificando que se ajustan a los objetivos deseados, obtienen los resultados correctos sin provocar fallos de ejecución. En la práctica, hablamos de un **proceso cíclico de desarrollo**: implementar, verificar, corregir, etcétera. En cierto momento, puede que tengamos que cambiar el diseño, e incluso puede que lo que esté fallando sea el modelo abstracto. Cuanto más atrás tengamos que volver, más tiempo habremos perdido haciendo cosas que luego resultan inútiles.

La evaluación de los programas incluye no sólo la corrección del resultado sino también la medición de las prestaciones, lo que normalmente se denomina el **análisis de eficiencia**. La eficiencia es una proporción entre los resultados obtenidos y los recursos consumidos. Fundamentalmente, un programa consume recursos de tiempo y de memoria. Aunque usualmente hablamos de "análisis de algoritmos", tanto las estructuras de datos como los algoritmos influyen en el consumo de recursos de un programa.

Incluso antes de llegar a la implementación de un programa, es posible y adecuado hacer el análisis del algoritmo en pseudocódigo. Este análisis será una previsión de la eficiencia que obtendría la solución diseñada si la implementamos. Se trata, por lo tanto, de un estudio teórico. Cuando el estudio se realiza sobre una implementación concreta, hablamos normalmente de estudios experimentales.

0.2. TIPOS DE DATOS

0.2.1. Definición de tipo de datos, tipo abstracto y estructura

Los conceptos de tipo abstracto de datos, tipo de datos y estructura de datos no son, en absoluto, redundantes ni incoherentes. Como ya hemos visto, aparecen en los distintos niveles de desarrollo. *Un tipo abstracto* es un concepto de diseño y por lo tanto previo e independiente de cualquier implementación. *Un tipo de datos* es un concepto de programación y se puede ver como la realización o materialización de un tipo abstracto. *La estructura de datos* es la organización o disposición en memoria de la información almacenada para cierto tipo de datos. Veamos, por partes, las definiciones.

Definición 0.1 ***Un Tipo Abstracto de Datos (TAD)*** *está compuesto por un dominio abstracto de valores y un conjunto de operaciones definidas sobre ese dominio, con un comportamiento específico.*

Por ejemplo, los booleanos constituyen un TAD, formado por el dominio de valores verdadero, falso y las operaciones sobre ese dominio: NO, Y, O, XOR, IMPLICA, etc. Decimos que el dominio de valores es abstracto porque no se le supone ninguna representación ni ningún significado. Es decir, el valor verdadero no tiene por qué ser un 1, ni debe almacenarse necesariamente en un bit. Pero, es más, ¿qué significa verdadero? Es un concepto hueco, vacío, sin ningún significado explícito. El único significado le viene dado implícitamente por su relación con las operaciones del tipo. Por ejemplo, verdadero Y cualquier cosa es siempre igual a cualquier cosa.

Las operaciones del TAD son abstractas en el sentido de que sabemos lo que hacen pero no cómo lo hacen. A la hora de programar el TAD, esto da lugar a un principio conocido como **ocultación de la implementación**: lo importante es que se calcule el resultado esperado y no cómo se calcule.

Definición 0.2 ***El tipo de datos*** *de una variable, un parámetro o una expresión determina el conjunto de valores que puede tomar esa variable, parámetro o expresión y las operaciones que se pueden aplicar sobre el mismo.*

El tipo de datos es un concepto de programación, de implementación, mientras que el TAD es un concepto matemático previo a la programación. Un tipo de datos debe ajustarse al comportamiento esperado de un TAD. Por ejemplo, el tipo integer de Pascal o el tipo int de C son dos tipos de datos, que corresponden al TAD de los enteros, Z. Además de los tipos de datos elementales existentes en un lenguaje de programación, los lenguajes suelen ofrecer mecanismos para que el usuario se defina sus propios tipos. Estos mecanismos pueden ser más o menos avanzados. En lenguajes como C o Pascal, la definición del tipo indica los atributos que se almacenan para las variables de ese tipo. Las operaciones se definen por separado.

En lenguajes orientados a objetos, como C++ o Java, los tipos definidos por el usuario se llaman clases, y su definición incluye tanto los atributos almacenados como las operaciones sobre los mismos.

En la programación de un tipo de datos surgen dos cuestiones: cómo almacenar en memoria los valores del dominio y cómo programar las operaciones del tipo de datos. La primera cuestión implica diseñar una estructura de datos para el tipo. La segunda está relacionada con la implementación de las operaciones de manipulación y, evidentemente, estará en función de la primera.

Definición 0.3 ***La estructura de datos de un tipo de datos*** *es la disposición en memoria de los datos necesarios para almacenar los valores de ese tipo.*

Por ejemplo, para representar las pilas podemos usar una estructura de arrays como la que aparece antes, o una lista enlazada de enteros, o una lista de arrays de enteros, o un array de listas de enteros, y así sucesivamente. Por lo tanto, un tipo de datos puede ser implementado utilizando diferentes estructuras.

De forma análoga, una misma estructura de datos puede corresponder a diferentes tipos. Por ejemplo, una estructura de arrays puede usarse para implementar listas, pilas o colas de tamaño limitado. Será la forma de actuar de las operaciones la que determine si el array almacena una lista, una pila o una cola.

La utilización de estructuras de datos adecuadas para cada tipo de datos es una cuestión fundamental en el desarrollo de programas. La elección de una estructura tendrá efectos no sólo en la memoria utilizada por el programa, sino también en el tiempo de ejecución de los algoritmos. Por ejemplo, la representación de colas con arrays puede producir desperdicio de memoria, ya que el tamaño del array debe ser el máximo previsible. Si se utilizan punteros se podrá usar menos memoria, pero posiblemente el tiempo de ejecución será sensiblemente mayor.

0.2.2. Tipos de tipos

Dentro de este apartado vamos a hacer un repaso de terminología, clases y propiedades relacionadas con los tipos de datos.

Clasificación de tipos según si el tipo es un tipo estándar del lenguaje o no.

- ***Tipos primitivos o elementales***. Son los que están definidos en el lenguaje de programación. Por ejemplo, en C lo son los tipos int, long, char y float, entre otros.
- ***Tipos definidos por el usuario***. Son todos los tipos no primitivos, definidos por la misma persona que los usa, o bien por otro programador.

Idealmente, un buen lenguaje debería hacer transparente el uso de tipos de una u otra categoría. Es decir, al usar una variable de cierto tipo T debería ser lo mismo que T sea un tipo primitivo o definido por el usuario. Ambos deberían ser conceptualmente equivalentes. Pero esto no siempre ocurre. Por ejemplo, en C es posible hacer asignaciones entre tipos primitivos, pero no entre ciertos tipos definidos por el usuario. La asignación "a= b;" funcionará o no, dependiendo de que los tipos de a y b sean primitivos o no.

Clasificación de tipos según el número de valores almacenados

- ***Tipo simple***. Una variable de un tipo simple contiene un único valor en cierto instante. Por ejemplo, los enteros, reales, caracteres y booleanos son tipos simples.
- ***Tipo compuesto***. Un tipo compuesto es aquel que se forma por la unión de varios tipos simples o compuestos. Por lo tanto, una variable de tipo compuesto contiene varios valores de tipo simple o compuesto.

Mecanismos de composición para crear tipos compuestos en los lenguajes de programación: ***arrays y registros***. Un array contiene una serie de posiciones consecutivas de variables del mismo tipo, cada una identificada con un índice dentro del rango del array. Los registros se definen enumerando los campos que forman el tipo compuesto, cada uno de los cuales tiene un nombre y un tipo (que, a su vez, puede ser simple o compuesto). Las clases, de lenguajes orientados a objetos, también se pueden ver como una manera de crear tipos compuestos, en la cual se pueden añadir atributos y operaciones al tipo.

Normalmente los tipos simples son tipos elementales y los tipos compuestos son definidos por el usuario. No obstante, también pueden existir tipos compuestos definidos en el lenguaje de programación, y tipos simples definidos por el usuario. Un mecanismo que permite al usuario definir nuevos tipos simples son los **enumerados**. Un enumerado se define listando de forma completa el dominio del tipo

Un tipo contenedor, o **colección**, es un tipo compuesto que puede almacenar un número indefinido de valores de otro tipo cualquiera. Por ejemplo, una lista de enteros o un array de Sexo son tipos contenedores. Un registro con dos enteros no sería un contenedor. En general, interesa que los tipos contenedores puedan almacenar valores de otro tipo cualquiera; por ejemplo, que las listas puedan ser de enteros, de reales, de registros o de cualquier otra cosa. Un tipo se dice que es **genérico o parametrizado** si su significado depende de otro tipo que es pasado como parámetro. Por ejemplo, una lista genérica tendría la forma Lista[T], donde T es un parámetro que indica el tipo de los objetos almacenados. Podríamos tener una variable a de tipo Lista[entero] y otra variable b de tipo Lista[real]; estos casos concretos se llaman **instanciaciones** del tipo genérico.

Realmente son muy pocos los lenguajes que permiten la definición de tipos parametrizados. Entre ellos se encuentra C++, que permite parametrizar un tipo mediante el uso de plantillas template. En la parte de prácticas se profundizará más en el uso de plantillas como una manera de crear tipos genéricos.

Finalmente, podemos hacer una distinción entre tipos mutables e inmutables. Decimos que un tipo de datos es **inmutable** cuando los valores del tipo no cambian después de haber sido creados. El tipo será **mutable** si los valores pueden cambiar. ¿Qué implicaciones tiene que un tipo sea mutable o inmutable? Supongamos que queremos definir el tipo Lista[T]. Si el tipo se define con una representación mutable, podemos incluir operaciones que añaden o suprimen elementos de la lista.

operación Inserta (var lista: Lista[T]; elemento: T)
operación SuprimePrimero (var lista: Lista[T])

donde *lista* es un parámetro que se puede modificar dentro de las operaciones. Pero si el tipo es inmutable no sería posible que un parámetro de entrada se modificara. En ese caso, las operaciones de inserción y eliminación deberían devolver una nueva lista, con el elemento correspondiente insertado o eliminado.

operación Inserta (lista: Lista[T]; elemento: T): Lista[T]
operación SuprimePrimero (lista: Lista[T]): Lista[T]

Según nos interese, definiremos los tipos como mutables o inmutables. Normalmente, la mayoría de los tipos serán mutables, aunque en algunos casos nos interesará usar tipos inmutables: la suposición de que los valores del tipo no cambian puede simplificar, o hacer más eficiente, la implementación de la estructura de datos para el tipo.

0.2.3. Repaso de tipos y pseudolenguaje de definición

La mayoría de los lenguajes ofrecen al programador un conjunto similar de tipos de datos simples (enteros, reales, booleanos, etc.), con las operaciones necesarias para manejarlos. Estos tipos tienen una correspondencia con los tipos manejados por el procesador, por lo que su implementación es muy eficiente.

- ***Enteros***. Con signo o sin signo, con precisión simple, doble o reducida (un byte).
- ***Reales***. Con precisión simple o doble. Para los números reales se suelen usar representaciones mantisa-exponente. Cuando lo usemos, pondremos el tipo real.
- ***Caracteres***. Lo denotaremos como caracter.
- ***Cadenas***. En algunos lenguajes las cadenas de caracteres son tipos elementales. En otros, como en C, las cadenas no son un tipo elemental sino que son simplemente arrays de caracteres.
- ***Booleanos***. Igual que antes, no existen en C –donde se usan enteros en su lugar–, aunque sí en C++ y Pascal. Supondremos que tenemos el tipo booleano.

En cuanto a los tipos contenedores (listas, pilas, colas, árboles, etc.), normalmente no forman parte del lenguaje de programación, sino que se definen en librerías de utilidades que se pueden incluir en un programa o no. Pero sí que se suelen incluir mecanismos de composición de tipos como arrays, registros y tipos enumerados. En nuestro pseudolenguaje tenemos la posibilidad de definir nuevos tipos, con una cláusula **tipo**. Y tb permitimos que se definan tipos parametrizados.

```
tipo
    DiasMeses = array [1..12] de entero
    Pila[T] = registro
        datos: array [1..MAXIMO] de T
        tope, maximo: entero
    fin
    Sexo = enumerado (masculino, femenino)
```

Para los arrays, se indica entre corchetes, [min..max], el rango mínimo y máximo de los índices del array. Para los registros, se listan sus atributos, cada uno con su nombre y tipo. Y, por otro lado, los enumerados se forman listando los valores del dominio. En algunos sitios usaremos también registros con variantes. Un registro con variantes es un registro que contiene un campo especial, en función del cual una variable podrá tener unos atributos u otros (pero nunca ambos a la vez).

```
tipo
    Empleado = registro
        nombre: cadena
        edad: entero
        según sexo: Sexo
            masculino: (curriculum: cadena; sueldo: entero)
            femenino: (estadoCivil: enumerado (soltera, otros); telefono: entero)
        fin
    fin
```

En cuanto a los tipos colecciones, además de los arrays daremos por supuesto que disponemos de los siguientes:

- ***Listas***. Tendremos listas parametrizadas, Lista[T], y que guardan una posición actual. De esta forma, la inserción, eliminación, etc., se hacen siempre sobre la posición actual. Por conveniencia, en algunos sitios no se darán por supuestas las listas.
- ***Pilas***. Una pila es una lista LIFO: last in, first out (último en entrar, primero en salir). Suponemos el tipo genérico Pila[T], con las operaciones push, pop y tope.
- ***Colas***. Una cola es una lista FIFO: first in, first out (primero en entrar, primero en salir). Suponemos el tipo genérico Cola[T], con las operaciones meter, sacar y cabeza.

Por último, pero no menos importante, tenemos el tipo de datos puntero. Considerado como un tipo non-grato por algunos autores por tratarse de un concepto cercano al bajo nivel, lo cierto es que los punteros son esenciales en la definición de estructuras de datos. Los punteros son un tipo parametrizado. Una variable de tipo Puntero[T] contiene una referencia, o dirección de memoria, donde se almacena una variable de tipo T.

En nuestro pseudolenguaje suponemos que tenemos el tipo Puntero[T] con las siguientes operaciones y valores especiales:

- ***Operador de indirección***. Si a es de tipo Puntero[T], entonces la variable apuntada por a se obtiene con el operador de indirección flecha, ↑, esto es: a↑ es de tipo T.
- ***Obtención de la dirección***. Si t es una variable de tipo T, entonces podemos obtener la dirección de esa variable con la función PunteroA(t: T), que devuelve un puntero de tipo Puntero[T].
- ***Puntero nulo***. Existe un valor especial del tipo Puntero[T], el valor predefinido NULO, que significa que el puntero no tiene una referencia válida o no ha sido inicializado.
- ***Creación de una nueva variable***. Para crear una nueva variable apuntada por un puntero, suponemos la función genérica Nuevo(T: Tipo), que devuelve un Puntero[T] que referencia a una nueva variable creada. Esta variable se borrará con la función Borrar(a: Puntero[T]).
- ***Comparación***. Suponemos que hay definidos operadores de comparación de igualdad y desigualdad entre punteros (=, ≠).

No consideramos otras operaciones sobre punteros, al contrario de lo que ocurre con C/C++, donde los punteros son consideramos como enteros, pudiendo aplicar sumas, restas, etc. Por esta razón, entre otras, decimos que lenguajes como C/C++ son lenguajes débilmente tipados: existen pocas restricciones en la asignación, mezcla y manipulación de tipos distintos. Como contraposición, los lenguajes fuertemente tipados, como Pascal, tienen reglas más estrictas en el sistema de compatibilidad de tipos. En la práctica, esta flexibilidad de C/C++ le proporciona una gran potencia pero es fuente de numerosos errores de programación.