Capítulo 1

Paradigmas de la computación

En este tema se presentan conceptos relacionados con los diferentes paradigmas de programación existentes. Según Louden [15]: "*así como la forma en que nos comunicamos influye en lo que pensamos y viceversa, la forma en que programamos influye en lo que entendemos por computación y viceversa*". Con lo que el estudio de los paradigmas de los lenguajes de programación es tan importante para el programador, como lo es dominar unos lenguajes concretos, ya que este conocimiento va a permitir saber qué lenguaje es el mas adecuado para cada tipo de escenario y cada problema a resolver.

Antes de la década de los 40 (del siglo XX), se programaba cableando, y es en dicha década cuando Von Neumann [24] plantea el uso de códigos para determinar las acciones de los ordenadores, evitando el cableado. A continuación se asignaron símbolos a los códigos de las instrucciones y a las localizaciones de memoria, naciendo el **lenguaje ensamblador**.

Pero el lenguaje ensamblador, de **bajo nivel de abstracción**, dependía de cada ordenador y era difícil de entender. Se fueron añadiendo al lenguaje construcciones con mayor nivel de abstracción como la **asignación**, los **bucles** (también llamados **ciclos**) o las **sentencias condicionales** y opciones, que ya son instrucciones independientes del ordenador, mas concisas y fáciles de comprender. Pero al principio los lenguajes seguían reflejando la **arquitectura Von Newman**: un área de memoria donde se almacenaban tanto a los programas como a los datos de los mismos, y por separado había una unidad de procesamiento que ejecutaba secuencialmente las instrucciones del programa en memoria. Los lenguajes estaban muy lejos de lo que ahora se entiende por un **lenguaje de programación de alto nivel**.

Los lenguajes modernos, aunque siguen conservando en esencia ese tipo de procesamiento, al aumentar el nivel de abstracción y utilizar nuevas arquitecturas en paralelo, se hacen independientes de la máquina y los programas sólo describen el procesamiento en general, en lugar de detallar todas las instrucciones que debe ejecutar la unidad de procesamiento.

Así, siguiendo a Louden [15]: "*un lenguaje de programación es un sistema denotacional para describir computaciones en una forma legible tanto para el ordenador como para el*

programador". Un lenguaje de programación es una notación especial para comunicarse con el ordenador y la **computación** incluye todo tipo de operaciones como por ejemplo la manipulación de datos, el procesamiento de texto o el almacenamiento y la recuperación de información. A veces los lenguajes se diseñan con un propósito concreto (como SQL para el mantenimiento de una base de datos), pero los mas interesantes desde el punto de vista de paradigma, son los **lenguajes de propósito general**. Y estos han de ser legibles por un ordenador, requisito que exige la existencia de una estructura del lenguaje que permita a un programa su traducción no ambigua y finita. En general, se restringe la notación de un lenguaje de programación a la notación formal de un lenguaje de contexto libre (lenguajes descritos por unas formas especiales de reglas en su gramática).

La evolución de los lenguajes de programación se ha organizado en cinco generaciones:

1. En la **primera generación** se incluyen los **lenguajes máquina**, en los que los datos y las operaciones sobre ellos se describen mediante ceros y unos. Son códigos o notaciones muy difíciles de entender por los programadores y cada procesador tiene el suyo propio . Por ejemplo, el byte `01111000` le dice al procesador Z80 de Zilog que copie en el registro A el contenido del registro B.

2. La **segunda generación** es la que incluye a los lenguajes ensambladores, cuya traducción a lenguaje máquina es muy sencilla, y aún hoy se utilizan para tareas muy específicas, como puede ser para programar `drivers` para dispositivos. Siguiendo con el ejemplo anterior, el byte `01111000` se representa mediante el **mnemónico** "`LD A,B`", que es más sencillo de recordar[1].

3. La **tercera generación** es la que incluye a los lenguajes de alto nivel como Pascal, Fortran, C o Java. Se denominan de alto nivel porque están muy alejados de la máquina pero muy cercanos al los programadores. Para su traducción a lenguaje máquina se necesitan compiladores o intérpretes. Surgen alrededor de los años 60 (del siglo XX), siendo los primeros Fortran, Lisp, Algol y Cobol.

4. La **cuarta generación** agrupa a **lenguajes de propósito específico**, como SQL, Natural, o el del paquete estadístico SPSS que permite manipular grandes cantidades de datos con fines estadísticos.

5. Por último, en la **quinta generación** se incluyen lenguajes que se utilizan, en primer lugar, en el área de la Inteligencia Artificial, con los que se especifica mas qué problema hay que resolver que cómo se resuelve dicho problema con una secuencia de acciones. De los primeros que se incluyen en este grupo es el lenguaje Prolog, aunque otros lenguajes funcionales como Haskell, también se clasifican como de quinta generación.

[1] Ya que `LD` es una abreviatura de `LoaD`.

Finalmente, indicar que para grandes desarrollos en los que intervienen varios programadores, un lenguaje de programación se convierte en una parte de un entorno de desarrollo de software, que obliga a utilizar una metodología de desarrollo que permita comprender el programa como un todo e identificar fácilmente qué efecto produciría un cambio local. Los entornos, por lo tanto, se convierten en un conjunto de herramientas para la escritura y traducción de los programas, para manipular los archivos del programa, registrar cambios y realizar pruebas y análisis. Los entornos de programación son el objeto de la ingeniería del software, que queda fuera del objetivo de este tema: los lenguajes de propósito general.

1.1 Abstracción en los lenguajes de programación

La **abstracción** en los lenguajes de programación se refiere a la **abstracción de los datos**, que resume sus propiedades y la **abstracción del control** que resume las propiedades de la transferencia de control, esto es, de la modificación de la estrategia de ejecución de un programa en una situación determinada (por ejemplo, los bucles, las sentencias condicionales o las llamadas a subprogramas). A su vez las abstracciones se clasifican en básicas, estructuradas y unitarias[2].

1.1.1 Abstracciones de Datos

A continuación se presentan las abstracciones de datos, las cuales se trataran con mayor profundidad en el capítulo 5.

- Las **abstracciones de datos básicas** se refieren a la representación interna de los datos de **tipo atómico**[3] que ofrece el lenguaje, junto a sus operaciones estándar (como las aritméticas para los datos numéricos o las del **Álgebra de Boole** para los valores booleanos). Otro tipo de abstracción básica es el uso de nombres simbólicos para referenciar las localizaciones de memoria que contienen los datos del programa. Esto se conoce con el nombre de **variable**. Las variables abstraen estas direcciones por medio de un nombre y un tipo de datos establecidos mediante una **declaración**, por ejemplo el siguiente código en Pascal:

```
var x: integer;
```

  está declarando una variable `x` de tipo entero, mientras que su equivalente en C sería:

```
int x;
```

[2] Aunque en este libro no se tratarán en profundidad las abstracciones unitarias.
[3] Que son aquellos que no pueden dividirse en elementos más sencillos.

- Las **abstracciones de datos estructuradas** son el mecanismo de abstracción para colecciones de datos. Una estructura típica es el **array** (también llamado vector o, en una desafortunada traducción, arreglo) que reúne datos como una secuencia de elementos. Por ejemplo la declaración en C:

```
int tabla[7];
```

  establece que la variable `tabla` es un array de 7 valores enteros. En muchos lenguajes se puede dar nombre también a los tipos de datos, mediante una **definición de tipo** como la siguiente en C:

```
typedef int Mitabla[7];
```

  que define un nuevo tipo `Mitabla` que es un array de 7 enteros. A estos tipos se les denomina **tipos estructurados**.

- Las **abstracciones de datos unitarias** se refieren a la agrupación, como una única unidad, de datos y operaciones sobre ellos. Introducen el concepto de **encapsulado de datos** u ocultación de información, mecanismo muy útil para reunir códigos relacionados entre sí en localizaciones específicas dentro del programa, ya sea en forma de archivos por separado o como estructuras del lenguaje separadas dentro de un archivo. Estas abstracciones unitarias se asocian a menudo con los **tipos abstractos de datos**, separando las operaciones que se pueden realizar con los valores de dicho tipo de datos (lo que se conoce como **interfaz**) de su implementación interna.

  Ejemplos son los **módulos** en Haskell (o ML) y los **paquetes** en Java (o Ada). Las **clases** de los **lenguajes orientados a objetos** son un mecanismo conceptualmente mas cercano a las abstracciones unitarias, pero también a las abstracciones estructuradas, ya que ofrecen un encapsulamiento de datos y tienen algunas características de los módulos o paquetes.

  Dos de las características mas importantes de las abstracciones de datos unitarias son la capacidad de **reutilización** de la misma abstracción en programas diferentes, a través de bibliotecas y su **interoperabilidad** o facilidad de combinación de abstracciones al proporcionar convenciones estándar para sus interfaces como CORBA (*Common Object Request Broker Architecture*), que es un estándar de interface independiente del lenguaje de programación aplicado a la estructura de clase.

1.1.2 Abstracciones de Control

- Las **abstracciones básicas de control** son sentencias individuales que permiten modificar (directa o indirectamente) el control del flujo de la ejecución de un programa. Como ejemplos de abstracciones de control básicas están la **sentencia de asignación**

o el **goto** de Fortran, que se encarga del proceso de cambiar la transferencia de control de una sentencia a otra parte dentro del programa. Por ejemplo en el programa en Fortran:

```
1 GOTO 10
2 estas líneas se saltan...
3 ...
4 ...y la ejecución continúa aquí
5 10 CONTINUE
```

el control salta de la línea 1 a la 5. Todas las sentencias que haya las líneas intermedias no se ejecutan. Actualmente las sentencias `goto` se consideran de muy baja abstracción y en los lenguajes modernos se encuentran solo de forma muy limitada, por su escasa fiabilidad.

- Las **abstracciones de control estructuradas** agrupan sentencias más simples para crear una estructura con un propósito común que permite gobernar la ejecución del programa. Ejemplos típicos son los **bucles** o las **sentencias condicionales**, como `if`, la sentencia `case` de Pascal o el `switch` de C. Por ejemplo en C:

```
1 if (x >= 0) {
2   numSoluciones = 2;
3   r1 = sqrt(x) ;
4   r2 = -r1 ;
5 } else {
6   numSoluciones = 0 ;
7 }
```

el grupo de sentencias de las líneas 2 a la 4 (encerradas entre llaves) se ejecutan como un único bloque si se cumple que `x>=0` y en otro caso se ejecuta la sentencia 6.

Otros lenguajes como Haskell, utilizan la sangría como sustitución de llaves para indicar anidamiento, como en la siguiente función:

```
1 raices x
2   | numSoluciones x == 0 -> []
3   | numSoluciones x == 2 -> [sqrt(x), -sqrt(x)]
```

Además las abstracciones de control estructuradas se pueden anidar unas dentro de otras. Por ejemplo en C:

```
1 if (x > 0) {
2   numSoluciones = 2;
3   r1 = sqrt(x) ;
```

```
4   r2 = -r1 ;
5 } else if (x == 0) {
6           numSoluciones = 1 ;
7           r1 = 0.0;
8         }
9         else numSoluciones = 0;
```

Otro mecanismo muy útil para estructurar el control es el **subprograma**. Necesita una **declaración**, con un nombre y un conjunto de acciones a realizar que se abstraen bajo dicho nombre. Esta declaración es similar a la declaración de variable y de tipo. En segundo lugar es necesario que el subprograma sea llamado o invocado en el punto en que las acciones deben ejecutarse. Una **llamada a un subprograma** es un mecanismo mas complejo que las sentencias condicionales o los bucles, puesto que requiere el almacenamiento del estado del programa en el punto de llamada en el entorno de ejecución. Típicos ejemplos de subprogramas son los **procedimientos** de Pascal o los **métodos** de Java.

Un mecanismo de abstracción muy cercano al de subprograma es el de **función**, que es un subprograma que devuelve un resultado tras ser invocado. De hecho en algunos lenguajes como C, los procedimientos se consideran como funciones nulas (que no devuelven un valor). La diferencia mas importante entre procedimientos y funciones es que las funciones se corresponden con la abstracción matemática de función, por lo que pueden entenderse independientemente del estado del entorno de ejecución. Las funciones constituyen la base de la programación funcional, que se estudiará en el capítulo 2.

A continuación se incluye un ejemplo en Ada que calcula el máximo común divisor de los enteros `u` y `v` (sus **parámetros**):

```
1 function gcd ( u, v: in integer) return integer is
2   y, t, z: integer;
3 begin
4   z := u; y := v;
5   loop
6     exit when y = 0;
7       t := y;
8       y :=  z mod y;
9       z := t;
10    end loop;
11  return z;
12 end gcd;
```

En el capítulo 6 se estudiarán las abstracciones de control estructuradas.

- Las **abstracciones de control de tipo unitario** permiten agrupar una colección de subprogramas como una unidad en sí misma e independiente del programa. De esta

forma, aislando partes del programa cuyo funcionamiento no es necesario conocer en detalle, se mejora la comprensión del mismo. Esencialmente son idénticas a las abstracciones de datos unitarias (y generalmente se implementan con **módulos** y **paquetes** al igual que aquellas). Simplemente varía el enfoque, que en esta ocasión se orienta más a las operaciones que a los datos. No obstante mantienen las propiedades de las abstracciones de datos unitarias como la **reutilización** mediante la creación de bibliotecas.

Un tipo de abstracción de control difícil de clasificar en alguno de los niveles anteriores es el de los **mecanismos de programación en paralelo**, que los lenguajes modernos suelen incluir. Java por ejemplo, contiene los mecanismos de **hilos** (trayectorias de control ejecutadas por separado dentro del entorno Java). Ada contiene el mecanismo de **tarea** para lo mismo aunque se puede clasificar como una abstracción unitaria, mientras que los hilos y los **procesos** de Java son clases y por lo tanto son abstracciones estructuradas.

Abstracción	**de Datos**	**de Control**
Básica	tipos atómicos variables	asignación goto
Estructurada	tipos estructurados	bucles condicionales subprogramas
Unitaria	módulos paquetes	

Tabla 1.1: Abstracciones en los Lenguajes de Programación

En la tabla 1.1 se pueden ver, a modo de resumen, las diferentes abstracciones de los lenguajes de programación.

Finalmente, indicar que si un lenguaje de programación sólo necesita describir computaciones, entonces sólo necesita mecanismos suficientes para describir todos los cálculos que puede llevar a cabo una **máquina de Turing**, puesto que cualquier máquina de Turing puede ejecutar cualquier cálculo conocido en un ordenador. Un lenguaje de este tipo se conoce como **lenguaje completo en Turing**, debe incluir variables enteras y aritméticas, así como la ejecución de sentencias de forma secuencial, incluyendo sentencias de asignación, condicionales (`if`) y bucles (`while`).

1.2 Introducción a los paradigmas de computación

Inicialmente los lenguajes de programación se basaron en el **modelo de computación Von Neumann**, que propuso que el programa se almacenara en la máquina antes de ejecutarse y a su vez en:

1. La ejecución secuencial de instrucciones.

2. El uso de variables para la representación de las posiciones de memoria.

3. El uso de la asignación para cambiar el valor de las variables.

Estos lenguajes se conocen como **lenguajes imperativos**, porque sus instrucciones representan órdenes. También se les ha denominado **procedurales**, aunque no tengan nada que ver con el concepto de abstracción de procedimiento.

La mayoría de los lenguajes de programación son imperativos, pero no es requisito que la computación sea una secuencia de instrucciones donde cada una opere sobre un dato (esto se conoce como **cuello de botella de Von Neumann**), sino que la computación puede ser paralela, actuar sobre diferentes datos simultáneamente, o ser no determinista e independiente del orden. Hay otras formas de describir la computación de forma independiente al modelo Von Neumann, por lo que los lenguajes imperativos se consideran un paradigma o patrón (conocido como **paradigma imperativo**) para otros lenguajes de programación.

Dos paradigmas diferentes al anterior, basados en abstracciones matemáticas, son el **paradigma funcional**, que usa la noción de función según se plantea en el **lambda cálculo**, y el **paradigma lógico** que se basa en la lógica simbólica. Permiten que tareas muy complejas se describan precisa y concisamente, facilitando la verificación de los programas (comprobar si el programa se ejecutará correctamente).

En alguna bibliografía se denomina **programación declarativa** al grupo formado por la programación funcional y la lógica, por la gran diferencia de sus modelos de computación con los del resto de lenguajes de programación. En estos, las propiedades se declaran y no se especifica la secuencia de su ejecución. También se les denomina **lenguajes de muy alto nivel** o de **quinta generación**.

Un cuarto paradigma es el de la **programación orientada a objetos**, que facilita la reutilización de programas y su ampliación, siendo mas *natural* la elaboración de código que se quiere ejecutar. Sin embargo de alguna manera este paradigma es también imperativo pues se basa en una ejecución secuencial sobre un conjunto cambiante de posiciones de memoria. La diferencia es que los programas están formados por pequeñas piezas de código, cuyas interacciones están controladas y se cambian fácilmente. En la práctica este tipo de programación tiene dificultad en predecir con precisión el comportamiento y determinar la corrección de los programas. Actualmente es un estándar ampliamente utilizado.

A continuación se introducen con algo mas de detalle los paradigmas de orientación a objetos, funcional y lógico, utilizando un mismo ejemplo (calcular el máximo común divisor de 18 y 8) para iniciar el conocimiento de sus similitudes y diferencias, aspecto muy interesante ya que, en general, los lenguajes de programación actuales no se pueden clasificar únicamente en un paradigma, pues suelen contener características de diferentes paradigmas.

1.2.1 Programación orientada a objetos

Este paradigma se basa en la idea de que un objeto se puede describir como una colección de posiciones de memoria junto con todas las operaciones que pueden cambiar los valores de dichas posiciones. Un ejemplo muy básico de objeto es una variable con operaciones de asignación de valor y de recogida de su valor.

En la mayoría de los **lenguajes orientados a objetos**, los objetos se agrupan en clases que representan a todos los que tienen las mismas propiedades. Estas clases se definen mediante declaraciones parecidas a las de los tipos estructurados en C o Pascal. Tras la declaración de una clase, se pueden crear objetos concretos a partir de la misma, mediante la instanciación de la clase.

Para implementar el ejemplo del máximo común divisor en orientación a objetos se necesita una operación sobre objetos de tipo entero (`gcd`) y como los enteros ordinarios en Java no son objetos reales (por cuestiones de eficiencia), hay que incluir a los enteros en una nueva clase[4] que defina el objeto entero con la operación de máximo común divisor:

```
1 public class IntWithGcd {
2   public IntWithGcd( int val ) { value = val; }
3   public int intVal() { return value; }
4   public int gcd ( int v ) {
5     int z = value;
6     int y = v;
7     while ( y != 0 ) {
8       int t = y;
9       y = z % y;
10      z = t;
11    }
12    return z;
13  }
14  private int value;
15 }
```

En este ejemplo se define la nueva clase mediante:

1. Un constructor en la línea 2 (con el mismo nombre que la clase, pero sin tipo de salida). Los constructores asignan memoria y aportan los valores iniciales para los datos del objeto. En este caso el constructor necesita un entero, que es el valor del objeto.

2. Un método de acceso a este valor (`intVal` en la línea 3).

[4] Dado que no es posible añadir métodos a clase `Integer` de Java. Por eso es necesario crear una clase completamente nueva.

3. El método `gcd` (definido en las líneas de la 4 a la 13), con un único valor entero, ya que el primer parámetro es el valor del objeto sobre el que se llama a `gcd`.
4. El entero `value` queda definido en la línea 14.

El constructor y los métodos se definen con acceso público, para que puedan ser llamados por los usuarios, mientras que los datos de la línea 14 son privados para que no sean accesibles desde el exterior. La clase `IntWithGcd` se utiliza definiendo un nombre de variable para contener un objeto de la clase: `IntWithGcd x;`.

Al principio la variable `x` no contiene la referencia a un objeto, por lo que hay que instanciarla con la sentencia:

```
x = new IntWithGcd(8);
```

A continuación se llamaría al método `gcd` mediante:

```
int y = x.gcd(18);
```

Y tras la ejecución de esta sentencia, la variable `y` contendrá el valor 2, que es el máximo común divisor de 18 y 8.

En este ejemplo, el objeto de datos contenido en `x`, está enfatizado al colocarlo en primer término de la llamada (en vez de utilizar `gcd(x,18)`) y al darle solo un parámetro a `gcd`.

1.2.2 Programación funcional

La computación en el paradigma funcional se fundamenta en la evaluación de funciones o en la aplicación de funciones a valores conocidos, por lo que también se denominan **lenguajes aplicativos**. El mecanismo básico es la evaluación de **funciones**, con las siguientes características:

- La transferencia de valores como parámetros de las funciones que se evalúan.
- La generación de resultados en forma de valores devueltos por las funciones.

Este proceso no involucra de ningún modo a la asignación de una variable a una posición de memoria, aspecto que le aleja de la programación orientada a objetos. Tampoco las operaciones repetitivas se representan por ciclos (que requieren de **variables de control** para su terminación), sino mediante las **funciones recursivas**, un mecanismo muy potente.

Que un lenguaje de programación funcional prescinda de las variables y de los ciclos, ofrece ventajas relacionadas con la verificación de los programas. Volviendo al ejemplo de calcular el máximo común divisor (`gcd`), dicha función en un lenguaje funcional como Haskell sería:

```
1 gcd u v
2   | v == 0 -> u
3   | otherwise -> gcd v (mod u v)
```

En la línea 1 se define la cabecera de la función `gcd` y sus dos **parámetros formales** `u` y `v`. En la línea 2 se comprueba si `v` es igual a 0, en cuyo caso se devuelve directamente el valor contenido en el parámetro `u`.

En otro caso, la línea 3 establece la recursión, llamando nuevamente a la función `gcd` con los parámetros `v` y el resto de dividir `u` entre `v` (`mod u v`).

Ahora, para calcular el máximo común divisor entre 18 y 8, se deberá evaluar la expresión:

```
gcd 18 8
```

que nos devolverá 2.

La programación funcional se estudiará con más detalle en el capítulo 2.

1.2.3 Programación lógica

En un lenguaje de programación lógica, un programa está formado por un conjunto de sentencias que describen lo que es *verdad* o *conocido* con respecto a un problema, en vez de indicar la secuencia de pasos que llevan al resultado. No necesita de abstracciones de control condicionales ni de ciclos ya que el control lo aporta el modelo de inferencia lógica que subyace.

La definición de máximo común divisor (`gcd`) es la siguiente:

- El `gcd` de `u` y `v` es `u` si `v` es 0.
- El `gcd` de `u` y `v` es el `gcd` de `v` y de `u mod v`, si `v` no es 0.

y puede programarse directamente en un lenguaje de **PROgramación LOGica** como es **Prolog**, con el predicado (que podrá ser verdad o falso) `gcd(U,V,X)`, que se entiende como "*es verdad que el gcd de U y V es X*":

```
1 gcd(U, 0, U).
2 gcd(U, V, X) :- not (V = 0),
3                 Y is U mod V,
4                 gcd(V, Y, X).
```

Así, para calcular el máximo común divisor entre 18 y 8, se deberá escribir la consulta PROLOG:

```
?- gcd(18,8,X).
```

que busca un valor que, asignado a `X`, haga cierta esa pregunta.

En Prolog un programa es un conjunto de sentencias, denominadas **cláusulas**, de la forma: `a :- b,c,d.` que es una afirmación que se entiende como "*`a` es cierto, o resoluble, si `b`, a continuación `c` y finalmente `d` son ciertos o resolubles en este orden*". A diferencia de las funciones en la programación funcional, Prolog requiere de variables para representar los valores de las *funciones*, aunque no representan tampoco posiciones de memoria. En Prolog las variables se distinguen sintácticamente de otros elementos del lenguaje (por ejemplo, empezando por mayúsculas).

El lenguaje Prolog ha mostrado su interés para la programación de problemas complejos, cuando el uso de la recursividad sea necesaria y cuando no se conozca cómo o cuáles son los pasos para calcular o alcanzar un resultado. Además, casi todos los entornos de programación Prolog disponen de formas de comunicarse con otros lenguajes de programación, lo que permite escoger el paradigma de programación mas adecuado para cada parte del programa (cálculos complejos, en lenguaje C o Java, usando los hilos de Java para interfaces, etc). Ha mostrado, junto con la programación funcional, toda su capacidad en problemas clásicos en el área de la Inteligencia Artificial.

El estudio de la Programación Lógica se realizará en mayor profundidad en el capítulo 3.

1.3 Descripción de los lenguajes de programación

Los lenguajes de programación deben describirse de manera formal, completa y precisa. Esta descripción ha de ser, además, independiente de la máquina y de la implementación. Para ello se utilizan habitualmente estándares aceptados universalmente, ya que de esta formalización dependen tanto el diseño del propio lenguaje de programación como la comprensión del comportamiento del programa escrito por los programadores. Sin embargo no todos los niveles de descripción de un lenguaje disponen de un estándar para ello.

Los elementos fundamentales para la definición de un lenguaje de programación son los siguientes:

- El **léxico** o conjunto de las "palabras" o unidades léxicas que son las cadenas de caracteres significativas del lenguaje, también denominados `tokens`.

  También son unidades léxicas los **identificadores**, los símbolos especiales de **operadores**, como "+" o "<=" y los **símbolos de puntuación** como el punto y coma o el punto.

  Por ejemplo, una sentencia condicional en C tendría como tokens las cadenas `if` y `else`.

- La **sintaxis** o estructura conlleva la descripción de los diferentes componentes del lenguaje y de sus combinaciones posibles. Para ello se utilizan las **gramáticas libres de contexto**, un estándar aceptado universalmente. Por ejemplo la sentencia `if` en el lenguaje C se define por:

```
<sentencia if> ::= if ( <expresion> ) <sentencia> [else <
    sentencia>]
```

- La **semántica** expresa los efectos de la ejecución en un contexto determinado. A veces esta definición interactúa con los significados de otros elementos del lenguaje, y por ello, la semántica es la parte mas difícil en la definición de un lenguaje. Siguiendo con el ejemplo del `if` del lenguaje C, y según Kernighan y Richie [9]:

  > *Una sentencia `if` es ejecutada, primero, evaluando su expresión, que debe ser de tipo aritmético o apuntador (incluyendo todos sus efectos colaterales), y si el resultado de la comparación de la expresión es cierta, entonces se ejecuta la sentencia que sigue a la expresión. Si existe una parte `else` y el resultado de la expresión no es cierto, entonces se ejecuta la sentencia que sigue al `else`.*

  Además es necesario comprobar la *seguridad* de las sentencias. En el ejemplo del `if`, ¿qué ocurre si la expresión no se evalúa correctamente a cierto o falso, porque haya un error de división por cero? La alternativa a esta definición *incompleta* es el uso de un método formal para describir la semántica. Pero en la bibliografía no existe uno aceptado, por lo que es poco habitual encontrarse con definiciones formales de la semántica de un lenguaje, aunque existen algunos formalismos que construyen el significado del las construcciones del lenguaje.

  Entre los sistemas de notación para definiciones semánticas formales se encuentran la **semántica operacional** (el significado de una construcción es una descripción de su ejecución en una máquina hipotética), la **denotacional** (que asigna objetos matemáticos a cada componente del lenguaje para que modele su significado) y la **axiomática** (que modela el significado con un conjunto de axiomas que describen a sus componentes junto con algún tipo de inferencia del significado).

1.3.1 Traducción de los programas para su ejecución

Para la ejecución de los programas escritos en un lenguaje de programación, es necesario disponer de un traductor, un programa que acepta como entrada los programas del lenguaje y los ejecuta o transforma en una forma adecuada para su ejecución (lenguaje máquina). En el primer caso al traductor se le denomina **intérprete** y en el segundo **compilador**.

En el caso del intérprete la ejecución de un programa se realiza en un paso: con los datos necesarios y el programa como entrada, el intérprete produce la ejecución del programa sobre esos datos. La compilación, por su parte, es un proceso de dos pasos: el programa original o **código fuente** de la entrada se convierte en un nuevo programa o **código objeto**, que es el que puede ser ejecutado (si ya está en lenguaje máquina) sobre los datos que se desee.

En general, el lenguaje del código objeto debe ser a su vez traducido por un **ensamblador** en un nuevo código objeto, que será **linkado** (o unido) con otros códigos objeto, cargado en localizaciones de memoria adecuadas y finalmente ejecutado. Incluso en ocasiones el lenguaje objetivo es a su vez otro lenguaje de programación, con lo que el proceso es mas complejo, aunque similar. Otro caso posible es aquel en el que un **pseudo-intérprete** no produce un programa objetivo, sino que traduce el programa fuente a un lenguaje intermedio que posteriormente es interpretado (por ejemplo el lenguaje Perl).

También en general, se desea que el traductor siga exactamente a la definición del lenguaje de programación, aunque el programador tiene que estar al tanto de las características tanto del lenguaje como del traductor. En ocasiones un lenguaje está definido por el comportamiento de su intérprete o compilador en particular, denominado **traductor definicional**, aunque sea una mala práctica.

Las fases que tanto un intérprete como un compilador deben llevar a cabo son:

1. Primero, un **analizador léxico** debe identificar los tokens del programa (palabras clave, constantes, identificadores, etc.), ya que inicialmente el programa se entiende como una secuencia de caracteres. En ocasiones, hay un preprocesamiento previo, para transformar el programa en una entrada correcta del analizador léxico.

2. A continuación, un **analizador sintáctico** o gramatical identifica las estructuras correctas que definen las secuencias de tokens.

3. Finalmente, un **analizador semántico** asigna el significado de forma suficiente para su ejecución o la obtención del programa objetivo.

Estas fases exigen el mantenimiento de un entorno o **ambiente de ejecución**[5], que administra el espacio de memoria para los datos del programa y registra el avance de la ejecución. En general, un intérprete administra él mismo el ambiente de ejecución, mientras que un compilador lo administra de forma indirecta, incluyendo en el código las operaciones necesarias.

Cualquier lenguaje de programación puede disponer de un intérprete y/o un compilador. En general los intérpretes disponen además de mecanismos interactivos para que el usuario manipule la entrada y salida, introduzca el programa en un terminal, recoja los resultados

[5] Se denomina ambiente de ejecución al enlace a posiciones de memoria de las variables globales o los subprogramas. Se estudiarán con más detenimiento en los capítulos 4 y 6.

de manera determinada, etc. Sin embargo, también en general, los intérpretes son menos eficientes que los compiladores ya que estos permiten la **optimización del código** en análisis previos a la ejecución del programa. Suele ser una opción de diseño si un compilador ejecuta una o varias fases (por ejemplo, el lenguaje C tiene esta característica).

Hay otro aspecto que influye en la selección de un intérprete o un compilador. Son las propiedades del lenguaje que pueden ser determinadas antes de su ejecución o **propiedades estáticas** y las que no, llamadas **propiedades dinámicas**. Las propiedades estáticas típicas son las relacionadas con el léxico y la sintaxis de un lenguaje de programación. En C o en Pascal, también son estáticos los tipos de datos de las variables. En un lenguaje que sólo tenga asignación estática, es decir una posición de memoria fija para las variables durante toda la ejecución, puede utilizar un **ambiente totalmente estático** y en caso contrario usar un **ambiente totalmente dinámico**. Sin embargo hay posiciones intermedias, como es el caso de un **ambiente basado en pilas** (como en C o Pascal) que tiene aspectos estáticos y dinámicos.

Históricamente, los lenguajes imperativos tienen mas propiedades estáticas y usan un ambiente estático administrado por un compilador, y los lenguajes declarativos usan un intérprete, aunque también dispongan de compiladores para obtener mayor eficiencia.

Un último aspecto a considerar con respecto a la traducción es el relacionado con la **recuperación de errores** que favorece la **fiabilidad**, una propiedad importante de un traductor. Durante la traducción, el traductor se encuentra errores que debe indicar mediante mensajes de error apropiados y que, dependiendo de la complejidad inherente al error, puede resolver o al menos recuperar para poder seguir adelante con la traducción.

Los errores se clasifican de acuerdo con la fase de traducción en que se encuentran:

1. Los **errores léxicos** ocurren durante la fase de análisis léxico y suelen estar limitados al uso de caracteres ilegibles (o no admitidos). Los errores ortográficos son difíciles de identificar. Por ejemplo si aparece `while`, no se puede conocer a priori si es el nombre de una variable o una mala escritura de `while`, pero este tipo de error sí lo encontrará el analizador sintáctico.

2. Los **errores sintácticos** se refieren a `tokens` que faltan en expresiones o expresiones mal formadas. Así, en el ejemplo anterior, en caso de encontrar `while`, el analizador sintáctico lo interpretará como un identificador, originando un error sintáctico si en ese punto del programa no se espera un identificador.

3. Los **errores semánticos** pueden ser estáticos, detectados antes de la ejecución, como tipos incompatibles o variables no declaradas, o errores dinámicos, detectados durante la ejecución como una división por cero o un subíndice fuera de rango.

4. Los **errores lógicos**, que son también cometidos por el programador, pero además producen un comportamiento erróneo o no deseable del programa. En el siguiente ejemplo en C, cuando `u` y `v` sean iguales a 1, se entrará en un bucle infinito:

```
x = u;
y = v;
while (y != 0) {
  t = y;
  y :=  x * y;
  x = t;
}
```

y sin embargo, el programa no tiene errores léxicos ni sintácticos, y es semánticamente correcto desde el punto de vista del lenguaje, aunque su ejecución no sea la deseada en algunos casos.

En ocasiones, en la descripción de los lenguajes se especifican qué errores deben ser detectados antes de la ejecución, cuáles pueden provocar un error en tiempo de ejecución y cuáles pueden pasar desapercibidos, pero el comportamiento preciso de un traductor respecto a un error no suele estar especificado. La **pragmática** de los lenguajes de programación se ocupa de aspectos como la especificación de mecanismos para activación o deshabilitación de opciones de optimización, depuración y otras facilidades pragmáticas, que suelen incluirse en los traductores.

1.4 Diseño de los lenguajes de programación

En los apartados anteriores se han tratado diferentes aspectos relacionados con los lenguajes de programación, entendidos como una herramienta para describir computaciones, donde prevalecen aspectos pragmáticos como son la propiedad de **legibilidad** tanto de la máquina como de los programadores. Así, el reto de un lenguaje es lograr la potencia, expresividad y comprensión que requiere la legibilidad del programador (ser humano) conservando la precisión y simplicidad necesarias para su traducción al lenguaje máquina. La legibilidad del programador es parcialmente proporcional a las capacidades de abstracción del lenguaje, como son los mecanismos para la **abstracción de datos** mediante estructuras de datos y para la **abstracción del control** de la ejecución. Por lo tanto, el objetivo principal de la abstracción en el diseño de los lenguajes de programación es el control de la **complejidad**.

A continuación se muestran características pragmáticas del diseño de un lenguaje de programación que deben ser conocidas tanto para diseñar nuevos lenguajes como para elegir el lenguaje de programación adecuado como herramienta de programación de proyectos. Hasta nuestros días, ningún lenguaje es adecuado para cualquier escenario o tarea, por lo que es necesario conocer de cada lenguaje qué características (o principios) lo diferencian y cuáles los hacen mas adecuados para determinados escenarios o tareas. El programador debe conocer los principios de diseño prioritarios durante el diseño de cada lenguaje de programación para saber elegir. Por ejemplo, hay varios lenguajes de alto nivel (tercera

generación), que a priori pueden utilizarse para desarrollar cualquier programa, pero ocurre que Java y PHP se utilizan fundamentalmente para aplicaciones web, y C no, aunque podría utilizarse (con mayor dificultad).

Históricamente el criterio principal en el diseño de lenguajes de programación era la **eficiencia en la ejecución**, en términos de los ordenadores existentes en cada momento. Por ejemplo Fortran pretendía ser compacto y con un código ejecutable eficiente. La capacidad de escritura que permite al programador la expresión clara, correcta, concisa, precisa y rápida quedaba en segundo plano. Cobol y Algol60 ya aparecen como un paso hacia la legibilidad con mecanismos como **estructuración en bloques** o la **recursión**. Por ejemplo, el algoritmo de `quicksort` se define por primera vez por C.A.R. Hoare en Algol60. Cobol (acrónimo de *COmmon Business-Oriented Language, Lenguaje Común Orientado a Negocios*) pretendía acercar la programación en sentencias y expresiones en inglés, con lo que los programas eran largos y demasiado verbosos. Por lo tanto se puede decir que ninguno de los dos lenguajes consiguió el objetivo de la legibilidad, pero fueron pioneros en tenerla en cuenta en la fase de diseño de los lenguajes de programación.

Simula67 y Algol68, son lenguajes que ya incorporaron mecanismos de abstracción. Algol68 se diseña totalmente general y ortogonal con pocas restricciones para que se pudiera hacer casi cualquier combinación significativa de las estructuras disponibles. Esto provocaba un incremento de la complejidad, ya que los constructores excesivamente generales son mas difíciles de comprender, sus efectos menos predecibles y la computación subyacente mucho mas oscura. Simula67 introdujo su concepto de clase en influyó en muchos lenguajes creados en los años 70 y 80 (del siglo XX). Sin embargo todavía la pretendida elegancia conceptual condujo a implementaciones ineficientes que los hacía inútiles en algunos contextos y problemas.

En los años 70 y 80 aparecen otros lenguajes que enfatizaban la **simplicidad** y la **abstracción** como Pascal, C, Euclid, CLU, Modula-2 y Ada. Incluyen definiciones matemáticas para definir a los constructores y mecanismos para facilitar la comprobación de los programas, con objeto de mejorar la confianza en los programas, esto es su fiabilidad. Como consecuencia de estos esfuerzos, disponer de un fuerte **sistema de tipos** se ha convertido en una parte estándar de la mayoría de los lenguajes de programación.

En los años 80 y 90 aumenta el interés por los lenguajes declarativos y se reaviva con la aparición de ML y Haskell (1999), aunque continúa la popularidad de Prolog y Lisp/Scheme.

Quizá la mayor evolución histórica en busca de un mayor nivel de abstracción se deba al diseño de lenguajes que facilitan la programación orientada a objetos, como Smalltalk, C++ y Java. En estos lenguajes el uso de bibliotecas para tareas específicas y de técnicas orientadas a objetos han incrementado además la flexibilidad y reutilización del código existente.

Por lo tanto, a través de los años se puede observar que el énfasis sobre diferentes objetivos de diseño se ha ido modificando, aunque los temas de legibilidad, abstracción y control de la complejidad son comunes a casi todos los diseños.

1.4.1 La eficiencia

Este principio se refiere a que el diseño debe permitir al traductor la generación de **código ejecutable eficiente**, también conocido como **optimizabilidad**. Dos ejemplos: las variables con tipos estáticos permiten generar código que las asigna y referencia con eficiencia. En C++ el diseño de clases, en ausencia de características orientadas a objetos avanzadas, no requiere de memoria o código adicional mas allá del mecanismo `struct`.

La eficiencia se organiza en tres principios: eficiencia de traducción, de implementación y de programación.

- La **eficiencia de traducción** estipula que el diseño del lenguaje debe permitir el desarrollo de un traductor eficiente y de tamaño razonable. Por ejemplo Pascal o en C exigen, por diseño, que se declaren las variables antes de su uso, permitiendo un compilador de una sola pasada. Esta restricción se ha liberado en C++, por lo que se requiere un compilador de dos pasadas en ciertas partes del código para resolver las referencias de los identificadores.

  Hay traductores que se diseñan para que los programadores omitan la verificación de errores con reglas muy difíciles de verificar en tiempo de traducción, lo que influye negativamente en otro principio de diseño: la **fiabilidad** que aparece ligada a la verificación de errores.

- La **eficiencia de implementación**, es la eficiencia con que se puede escribir un traductor, que a su vez depende de la complejidad del lenguaje. Casos de fracaso son:

  - Algol60, porque la estructura basada en pilas necesaria para su ambiente de ejecución no era suficientemente conocida en aquel momento.
  - En ML la inferencia de tipos sin declaraciones tuvo que esperar a la aplicación del algoritmo de **unificación** para que se utilizara de forma comprensible.
  - El tamaño y la complejidad de Ada era un obstáculo para el desarrollo de compiladores, dificultando de esta forma su disponibilidad y su uso.

- La **eficiencia de la programación** está relacionada con la rapidez y la facilidad para escribir programas o **capacidad expresiva** del lenguaje. La capacidad expresiva de un lenguaje se refiere a la facilidad para escribir procesos complejos de forma que el programador relacione de manera sencilla su idea con el código. Es un aspecto relacionado con la potencia y la generalidad de los mecanismos de abstracción y la sintaxis. Desde este punto de vista LISP y Prolog son lenguajes ideales con una sintaxis concisa, ausencia de declaración de variables e independencia con el mecanismo de ejecución, aunque compromete otros principios como son la legibilidad, la eficiencia en la ejecución y la fiabilidad.

  Relacionado con este concepto, se encuentra el de **Azúcar Sintáctico**, término acuñado por Peter Landin en 1964 [13] para describir aquellas estructuras sintácticas que

no añaden expresividad al lenguaje, pero que facilitan la escritura de los programas ofreciendo alternativas para que el programador pueda escoger. Un ejemplo de azúcar sintáctico se puede encontrar en las diferentes versiones de **bucles** que ofrecen los lenguajes de programación.

Finalmente indicar que la fiabilidad (un programa es fiable si hace lo que se espera y si permite al programador recuperar errores), exige de tiempo adicional para pruebas, recuperación de errores o agregación de nuevas características (**capacidad de mantenimiento**) y depende tanto de los recursos disponibles como de su consumo. Es un problema del que se encarga la ingeniería del software. Por lo tanto, la eficiencia de crear software depende mas de la legibilidad y capacidad de mantenimiento que de la facilidad de escritura.

1.4.2 La regularidad

La **regularidad** de un lenguaje de programación es un principio que se refiere al comportamiento de las características del lenguaje. Se subdivide en tres propiedades: la generalidad, la ortogonalidad y la uniformidad, y si se viola una de ellas, el lenguaje ya se puede clasificar como irregular. En general, las irregularidades durante el diseño de un lenguaje son por causa de las prioridades de diseño, como por ejemplo, las irregularidades cometidas en C++ para conseguir su compatibilidad con C (objetivo prioritario). Sin embargo si una irregularidad no puede justificarse (por contravenir un principio de diseño prioritario) entonces probablemente sea un error de diseño.

- La **generalidad** se consigue cuando el uso y la disponibilidad de los constructores no están sujetas a casos especiales y cuando el lenguaje incluye solo a los constructores necesarios y el resto se obtienen por combinaciones de constructores relacionados. A continuación se incluyen algunos ejemplos:
  - Pascal admite funciones y procedimientos anidados, pueden ser parametrizados, pero no existen variables de procedimientos, por lo que los procedimientos carecen de generalidad. C carece de funciones o procedimientos anidados, por lo que los procedimientos también carecen de generalidad. Sin embargo, la mayoría de los lenguajes funcionales como Haskell, tienen un constructor de función completamente general.
  - Pascal no tiene arrays de longitud variable con lo que carecen de generalidad. C y Ada sí tienen arrays de longitud variable. En C no se pueden comparar dos arrays utilizando el operador de igualdad ==, deben ser comparados elemento a elemento, por lo que es el operador de igualdad el que carece de generalidad.
  - Muchos lenguajes no tienen mecanismos para extender el uso de operadores predefinidos, sin embargo Haskell sí, e incluso permite que se creen nuevos operadores por parte del usuario (a diferencia de Ada y C++), luego sus operadores han logrado una generalidad completa.

- La **ortogonalidad** (o independencia) ocurre cuando los constructores del lenguaje pueden admitir combinaciones significativas y en ellas, la interacción entre los constructores o con el contexto, no provocan restricciones ni comportamientos inesperados. Esto implica que los constructores del lenguaje no deben comportarse de manera diferente en contextos diferentes, por lo que las restricciones que dependen del contexto no son ortogonales. El mejor ejemplo de lenguaje ortogonal fue Algol68, en el que todos los constructores pueden combinarse en todas las formas significativas posibles. A continuación se incluyen otros ejemplos:

  - La no generalidad de comparación para la igualdad en C (mostrada en la propiedad anterior), puede considerarse como una no ortogonalidad, puesto que la aplicación de la igualdad depende de los tipos de valores que se están comparando y por lo tanto del contexto.
  - En Pascal las funciones solo pueden devolver valores de tipo escalar o array, y en C o C++ las funciones pueden devolver cualquier tipo de datos, excepto arrays o matrices (que son tipos tratados de forma diferente al resto). En Ada y en los lenguajes funcionales esta no ortogonalidad se elimina.
  - En C las variables locales solo se pueden definir al principio del bloque, y esta no ortogonalidad es eliminada en C++, en el que las variables se pueden definir en cualquier punto del bloque antes de su utilización. En C hay otra no ortogonalidad al pasar todos los parámetros por valor, excepto los arrays que se pasan por referencia.

- La **uniformidad** se refiere a que *lo similar se ve similar y lo diferente, diferente* lo que implica la consistencia entre la apariencia y el comportamiento de los constructores. Algunos ejemplos de no uniformidad son:

  - En C++ es necesario el punto y coma después de la definición de clase, pero está prohibido detrás de la definición de una función.
  - En C los operadores `&` y `&&` parecen similares pero su comportamiento es muy diferente, el primero es la conjunción bit a bit y el segundo la conjunción lógica.

  Finalmente indicar que las no uniformidades también pueden considerarse no ortogonalidades pues ocurren en contextos particulares y pueden considerarse interacciones entre constructores.

1.4.3 Principios adicionales

En los apartados siguientes, se presentan algunos principios adicionales a la legibilidad, eficiencia y regularidad del diseño de lenguajes de programación, que ayudan a definir un buen diseño o a elegir el mejor lenguaje para un escenario o unas tareas concretas.

1. **Simplicidad**

   Es un principio que se refiere sintácticamente a que cada concepto del lenguaje se presente de una forma única y legible y semánticamente que contiene el menos número posible de conceptos y estructuras con reglas de combinación sencillas. No debe confundirse con regularidad, por ejemplo Algol68 es regular pero no simple. Que haya pocos constructores puede ayudar, pero no siempre. Por ejemplo los lenguajes declarativos tienen pocos constructores básicos, pero un ambiente de ejecución muy complejo.

   Por otra parte, Basic es simple, pero carece de algunos constructores básicos como son las declaraciones y los bloques, lo que hace difícil programar aplicaciones grandes. También la simplicidad es una característica del diseño del lenguaje C, aunque en realidad es consecuencia de otro principio prioritario en C, el conseguir código objetivo eficiente y compiladores pequeños.

   La sobresimplicidad de Pascal ha influido en muchas de sus no uniformidades, como en el caso del uso de la asignación para el retorno de funciones (que es confusa), y ha provocado otras ausencias como la compilación por partes o de buenas herramientas de entrada y salida. De hecho la sobresimplicidad fue el motivo de su reemplazo por C, C++ y Java. A su vez, el éxito de C en alcanzar simplicidad, tuvo como consecuencia errores importantes: una sintaxis de tipos y operadores oscura, una semántica extraña para los arrays y una verificación de tipos débil.

2. **Expresividad**

   Es la facilidad con la que un lenguaje de programación permite expresar procesos y estructuras complejas. Uno de los mecanismos mas expresivos es la recursividad (Algol60 y lenguajes declarativos fueron los primeros en incluirla). En los lenguajes declarativos tanto los datos como el programa pueden cambiar durante su ejecución, aspecto muy útil en situaciones complejas, como cuando el tamaño o los datos pueden ser desconocidos inicialmente. Pero una expresividad muy alta entra en conflicto por ejemplo, con la simplicidad ya que el ambiente de ejecución es complejo.

   Los lenguajes orientados a objetos son también muy expresivos y han mostrado su utilidad para que los programadores escriban código complejo. El creador de C++ Stroustrup [22] indica que lo diseñó tras trabajar en Simula67 y su concepto de clase. Sin embargo esta expresividad que a veces se relaciona con la concisión, entra en conflicto con la legibilidad, como se muestra en el código siguiente, para copiar una cadena en otra:

```
while (*s++ = *t++);
```

3. **Extensibilidad**

   Es la propiedad relacionada con la posibilidad de añadir nuevas características a un lenguaje, como nuevos tipos de datos o nuevas funciones a la biblioteca. O también

añadir palabras clave y constructores al traductor como en los lenguajes declarativos, que tienen pocas primitivas y en los que se van incorporando constructores en el ambiente de ejecución, lo que es difícil de realizar en un lenguaje imperativo. La extensibilidad es una propiedad del diseño que actualmente se considera prioritaria. La tendencia es permitir que el usuario defina nuevos tipos de datos, nuevas operaciones sobre ellos y que su tratamiento sea como si se hubieran definido en el lenguaje desde el principio.

Es muy útil para definir nuevos tipos de datos, como el de matrices, y que las operaciones entre matrices se puedan escribir de forma similar a las operaciones entre enteros (por ejemplo `C := A + B;`, siendo `A`, `B` y `C` matrices). En este caso, se dice que la operación `+` está **sobrecargada**. En C++ y Ada la sobrecarga de operadores está limitada a operadores existentes y debe incluir las propiedades sintácticas de los mismos (por ejemplo para el operador `+`, la asociatividad a la izquierda). En Haskell también pueden agregarse operadores definidos por el usuario, como en el siguiente caso, en que se indica que un nuevo **operador** `+++` debe usarse como infijo (escrito entre sus dos operandos) y con **asociatividad** a la derecha (indicado por la `r` de `infixr`) y un nivel de **precedencia** 6 (el mismo que el signo `+` en el lenguaje).

```
infixr 6 +++
```

Otro aspecto de extensibilidad importante es la **modularidad**, es decir, la capacidad de disponer de bibliotecas y agregar nuevas, la práctica habitual es disponer de una considerada como el núcleo del lenguaje y otras externas, que pueden ser estándar (presentes en todas las implementaciones del lenguaje) o no. Ada por ejemplo, incluye las características de programación concurrente directamente en la biblioteca del núcleo (mecanismo de tareas), Java las coloca en una biblioteca estándar (la biblioteca hilos), mientras que C++ no especifica ninguna característica en particular sobre la concurrencia.

La modularidad se corresponde también con la posibilidad de dividir un programa en partes independientes (denominadas módulos, paquetes o unidades) que puedan enlazarse para su ejecución. Estos módulos pueden ser reutilizados o cambiados sin que afecten al al resto del programa (**escalabilidad**). Además es necesario que tengan interfaces bien definidas para que interactúen entre ellos. Es el caso de las APIs[6].

4. **Capacidad de restricción**

   Ante el tamaño y complejidad crecientes de los lenguajes como C++ y Java, la capacidad de restricción es una propiedad tan importante como la extensibilidad.

   La capacidad de restricción se refiere a la posibilidad de que un programador utilice solo un subconjunto de constructores mínimo y por lo tanto solo necesite un

[6] Del inglés "*Application Programming Interface*" ("*Interfaz de Programación de Aplicaciones*"), se refiere a la forma de comunicarse desde un programa con el conjunto de subprogramas que ofrece una biblioteca.

conocimiento parcial del lenguaje. Esto ofrece dos ventajas: el programador no necesita aprender todo el lenguaje y, por otra parte, el traductor puede implementar sólo el subconjunto determinado porque la implementación para todo el lenguaje sea muy costosa e innecesaria. Sin embargo en este caso surge la siguiente pregunta: ¿por qué no hacer que el lenguaje sea sólo ese subconjunto y el resto forme parte de bibliotecas?

Hay varias respuestas a la pregunta anterior. Si el manejo de la concurrencia sea de importancia sólo para algunas aplicaciones, las herramientas necesarias para su control podrían pertenecer a una biblioteca. Pero, por otra parte, en ocasiones es muy útil disponer de varias versiones del mismo tipo de constructores, como en el caso de las sentencias condicionales o los ciclos `while`, por su mayor expresividad en diferentes circunstancias. Esto último resulta innecesario estrictamente, pero puede ayudar a la comprensión de un programa, entrando en conflicto con la complejidad del lenguaje, que aumenta, al haber varias formas de hacer una misma cosa y haciendo de este modo decrecer la expresividad.

Otro aspecto relacionado con la capacidad de restricción es la **eficiencia**: porque un programa no utilice ciertas características del lenguaje, su ejecución no debe ser mas ineficiente. Por ejemplo un programa en C++ que no utilice el manejo de excepciones, no debe ejecutarse mas lentamente que un programa equivalente en C, que no dispone de manejo de excepciones, simplemente porque en C++ sí esté disponible.

Otra ventaja que permite la capacidad de restricción o la existencia de subconjuntos del lenguaje es el **desarrollo incremental** del mismo. Por ejemplo Java ha evolucionado con respecto a la versión inicial. En cada nueva versión se han incluido características que no estaban en versiones anteriores, aunque los programas escritos en una versión son compatibles con programas de versiones anteriores y pueden utilizar su código.

5. **Consistencia entre la notación y las convenciones**

   Un lenguaje debe incorporar notaciones y cualquier otra característica que ya se hayan convertido en estándares, como lo son el concepto de programa, funciones y variable. Si estos aspectos estándar quedan perfectamente reconocibles, se facilita a los programadores experimentados el uso del lenguaje. Lo mismo ocurre con las notaciones relacionadas con las matemáticas, por ejemplo la de los operadores aritméticos (`+`, `-`, etc), notación se debe basar en el conocimiento existente.

   Otro ejemplo de consistencia es la utilización de la forma estándar de la sentencia `if-then-else` y otras sentencias de control. Un ejemplo negativo se encuentra en Algol68, que viola este principio al usar `mode` para declarar un nuevo tipo en lugar de `type` (ya considerado un estándar).

   Finalmente, indicar otra convención aceptada, como es el uso de espacios en blanco o de las líneas en blanco. Fortran los ignora y el ejemplo que sigue es ya famoso por

comprometer la legibilidad y la seguridad del lenguaje, ya que su ejecución provoca que se asigne el valor 1.10 a la variable `DO99Y`:

```
Do 99 Y = 1.10
```

6. **Precisión**

   Es la propiedad que exige una definición precisa del lenguaje, de forma que el comportamiento de los programas sea predecible. La precisión del lenguaje ayuda tanto a la fiabilidad de los programas, como a la confianza en los traductores, porque el programa se comporta igual en cualquier máquina (**portabilidad**).

   El uso de un estándar como el ISO (*International Organization for Standars*) durante el diseño y la existencia de un manual de referencia del lenguaje aumenta su precisión y portabilidad. Para la mayoría de los lenguajes existen estándares publicados: Lisp, Fortran, Ada, Pascal, Cobol, C, C++, Java etc.

   Los manuales de referencia, además, deben ser comprensibles por la mayoría, y es conocido el caso de Algol68, que para obtener mayor precisión, los diseñadores inventaron muchos términos haciendo mas difícil su comprensión y dificultando su aceptación.

7. **Portabilidad**

   La portabilidad se consigue si la definición del lenguaje de programación es independiente de una máquina en particular. Normalmente, los lenguajes interpretados o aquellos cuya ejecución se delega en una máquina virtual (como es el caso de Java) lo son.

   Para alcanzar este principio ayuda, por ejemplo, que el uso de tipos de datos predefinidos no asigne posiciones de memoria ni involucre otros aspectos de la máquina. Sin embargo, en ocasiones la independencia no es posible como es el caso de los números reales, que pueden necesitar una precisión infinita para una especificación exacta, mientras que la máquina solo soporta precisiones finitas. Una solución es la de Ada, que conocido el problema, aporta herramientas para especificar la precisión de los números y sus operaciones dentro del programa y así eliminar su dependencia de una máquina.

8. **Seguridad**

   Este principio pretende evitar los errores de programación, y permitir su descubrimiento. Por lo tanto la seguridad esta muy relacionada con la fiabilidad y la precisión. Sin embargo, la seguridad compromete a la capacidad expresiva del lenguaje y a su concisión, pues se apoya en que el programador especifique todo lo que sea posible en el código (para evitar errores).

Es un aspecto que depende mucho de las necesidades de los programadores. Por ejemplo los programadores de lenguajes declarativos (funcionales o lógicos), entienden como negativas tanto la verificación estática de tipos, como las declaraciones de variables. Sin embargo los programadores de otras aplicaciones no complejas conceptualmente, sino complejas por la cantidad de datos, cálculos o interacciones en un entorno distribuido, prefieren incluso más mecanismos de seguridad.

El equilibrio entre seguridad por una parte y expresividad y generalidad por otra, hacen difícil encontrar una solución acertada para todas las necesidades. Un caso positivo es Haskell (o ML) que siendo un lenguaje funcional, permite objetos de múltiples tipos, no requiere declaraciones de variables y sin embargo dispone de verificación estática de tipos.

9. **Interoperabilidad**

   La interoperabilidad se refiere a la facilidad que tienen diferentes tipos de ordenadores, redes, sistemas operativos, aplicaciones o sistemas para trabajar conjuntamente de manera efectiva, sin comunicaciones previas, para intercambiar información útil y con sentido (según el glosario de la *Dublin Core Metadata Initiative*). La organización para la web *W3C* define la interoperabilidad como la capacidad de un sistema de trabajar en conjunción con otros sistemas sin un esfuerzo especial por parte del usuario.

   Hay **interoperabilidad semántica**, cuando los sistemas intercambian mensajes entre sí, interpretando el significado y el contexto de los datos, **interoperabilidad sintáctica**, cuando un sistema lee datos de otro, mediante una representación compatible (para ello hay metalenguajes como el XML), e **interoperabilidad estructural** cuando los sistemas (con mo-delos lógicos compartidos) pueden comunicarse e interactuar en ambientes heterogéneos.

   Para hacer posible la interoperabilidad entre aplicaciones desarrolladas en diferentes lenguajes de programación y ejecutadas en cualquier plataforma, se debe establecer la representación de la información a intercambiar y además utilizar un protocolo de comunicación. Los servicios *web* son protocolos y estándares para intercambiar datos entre aplicaciones.

   La interoperabilidad entre lenguajes es la posibilidad de que un determinado código interactúe con otro escrito en un lenguaje de programación diferente. La primera dificultad aparece cuando los dos códigos se han desarrollado de manera independiente sin ningún requisito inicial de interoperabilidad. Un ejemplo de interoperabilidad a nivel de lenguaje es la **máquina virtual de Java**, que interpreta y ejecuta instrucciones en código intermedio Java (*bitecode*) sobre diferentes plataformas. Además de programas Java compilados a *bitecode*, los programas también se pueden compilar desde otros lenguajes de programación como Ada, Perl, Smalltalk, y otros.

   Por último, indicar que en ocasiones se puede escribir dentro de un programa, código en otro lenguaje de programación, con objeto de simplificar la programación. Un

ejemplo es incluir o *embeber* código SQL en un programa en otro lenguaje (*anfitrión*). Cuando se ejecuta el programa el intercambio de valores se realiza mediante variables (*huéspedes*), luego el resultado de la consulta SQL se asignaría a las variables definidas para ello.

1.5 Ejercicios resueltos

Aunque los ejercicios posibles de este tema son de carácter teórico fundamentalmente, algunas preguntas que pueden ayudar a reflexionar al lector son:

1. ¿En qué se diferencia la programación declarativa de la programación orientada a objetos?

   Solución:

   En la programación declarativa se especifica el conjunto de reglas, condiciones, afirmaciones o restricciones que describen el problema y que muestran el tipo de solución, sin necesitar especificar qué pasos hay que dar para llegar a dicha solución.

   En un lenguaje orientado a objetos, se utilizan los objetos para diseñar los programas, incluyendo herencia, cohesión, polimorfismo, abstracción, acoplamiento y encapsulamiento. Sobre estos objetos se deben especificar, además, todos los pasos necesarios para llegar a la solución del problema.

2. ¿Cree que afecta la capacidad de expresión de un lenguaje en la eficiencia de programación en el mismo?

   Solución:

   Sí, porque cuanto más naturales sean las expresiones permitidas en un lenguaje de programación, más fácil será programar y más eficiente serán los programas complejos. Una mayor capacidad de expresión de un lenguaje de programación favorece también el mantenimiento del software desarrollado (facilidad para corregir errores y añadir nuevas características).

1.6 Ejercicios propuestos

1. Defina y aporte un ejemplo de cada uno de los tres principios en que se divide la eficiencia de un lenguaje de programación.

2. Defina las tres propiedades en que se subdivide el principio de regularidad de un lenguaje de programación.

3. Describa brevemente e incluya un ejemplo de las siguientes características que afectan al diseño de un lenguaje respecto a la eficiencia:

(a) Eficiencia de código.

(b) Eficiencia de traducción.

(c) Eficiencia de programación.

(d) Eficiencia de implementación.

1.7 Notas bibliográficas

Para la elaboración de este capítulo se han consultado diferentes fuentes, pero sobre todo el libro [15], del que se han tomado algunos ejemplos paradigmáticos. También se han consultado [24], [13], [22] y [9].