Instituto de Ciencias de la Educación
POLITÉCNICA

Introducción a la programación funcional

Santiago Higuera de Frutos
Universidad Politécnica de Madrid

(Versión de fecha 19 de mayo de 2024)

A los alumnos y profesores de la Universidad Politécnica de Madrid y al oficio de profesor, que tan pocos reconocimientos recibe, a pesar de su importancia en la creación de una sociedad mejor.
Santiago

CONTENIDO

Prólogo XI

1 Pensar de manera funcional. Programación declarativa 1
1.1 Adiós a la Programación Orientada a Objetos . . . 2
1.2 ¿En qué consiste la programación funcional? . . . 9
1.3 Diferencias con la Programación Orientada a Objetos . . . 12

2 "¡Hola, Mundo!" en Rust 15
2.1 Instalación de Rust . . . 16
2.2 ¡Hola, mundo! en Rust . . . 20
2.3 ¡Hola, mundo!, utilizando Cargo . . . 22
2.4 Código de los ejemplos . . . 25
2.5 Conceptos comunes en programación . . . 26
2.6 Comentarios en los programas . . . 28

3 Elementos fundamentales del lenguaje Rust 29
3.1 Tipos de datos primitivos . . . 30
3.2 El tipo unidad . . . 34
3.3 Tipos de datos compuestos . . . 34
3.4 Tipos de datos personalizados . . . 36
3.5 Cadenas de caracteres . . . 41
3.6 Criterios para los nombres de los identificadores . . . 42
3.7 Mutabilidad . . . 43
3.8 Obtener el tipo de una variable . . . 44
3.9 El operador de asignación . . . 44
3.10 Otros operadores . . . 46
3.11 Funciones de los tipos en coma flotante . . . 50
3.12 Bifurcaciones . . . 52
3.13 Bucles . . . 55
3.14 Funciones . . . 58
3.15 *Traits* . . . 65

4 Propiedad de los valores 71
4.1 El concepto de propiedad de los valores . . . 72
4.2 Referencias . . . 75

5 Funciones de orden superior, closures e iteradores 77
5.1 Funciones de primera clase . . . 77
5.2 Errores de ejecución . . . 82
5.3 El *enum Option* . . . 83
5.4 Gestión de errores con *Result* . . . 86
5.5 El operador *?* . . . 88

5.6 Iteradores . . . 88
5.7 El método *map()* . . . 92
5.8 El método *filter()* . . . 94
5.9 El método *fold()* . . . 95
5.10 Herramientas de los Iteradores . . . 98

6 Funciones puras, inmutabilidad 101
6.1 Distinguir entre datos, cálculos y acciones . . . 102
6.2 Tipos de entradas y salidas de las funciones . . . 107
6.3 Separar las cosas . . . 107
6.4 Extraer los cálculos de las acciones . . . 108
6.5 Ejemplo: eliminar entradas implícitas . . . 110
6.6 Inmutabilidad. Copy-on-write . . . 112

7 Colecciones 115

8 Patrones de diseño en la Programación Funcional 117

9 Grokking Simplicity 119
9.1 Welcome to Grokking simplicity . . . 120
9.2 Functional thinking in action . . . 120
9.3 Distinguishing actions, calculations and data . . . 120
9.4 Distinción entre acciones, cálculos y datos . . . 123
9.5 Ejemplo . . . 125
9.6 Extracting calculations from actions . . . 126
9.7 Improving the design of actions . . . 126
9.8 Staying inmutable in a mutable language . . . 126
9.9 Staying inmutable with untrusted code . . . 126
9.10 Stratified design (I) . . . 126
9.11 Stratified design (II) . . . 126
9.12 First-class functions (I) . . . 126
9.13 First-class functions (II) . . . 126
9.14 Functional iteration . . . 126
9.15 Chaining functional tools . . . 126
9.16 Functional tools for nested data . . . 126
9.17 Isolating timelines . . . 126
9.18 Sharing resources between timelines . . . 126
9.19 Coordinating timelines . . . 126
9.20 Reactive and onion architectures . . . 126
9.21 The functional journey ahead . . . 126

10 Functional Programming made easier (Scalfani) 127
10.1 1.- Discipline is freedom . . . 127
10.2 Variables globales . . . 129

11 Railway Oriented Programming **135**
11.1 Monoides . . . 135

12 NaN an Inf **137**
12.1 Artículos para recomendar a los alumnos . . . 137

13 Funciones de orden superior, closures e iteradores **139**
13.1 Funciones de primera clase . . . 139
13.2 Errores de ejecución . . . 144
13.3 El *enum Option* . . . 145
13.4 Gestión de errores con *Result* . . . 148
13.5 El operador *?* . . . 150
13.6 Iteradores . . . 150
13.7 El método *map()* . . . 154
13.8 El método *filter()* . . . 156
13.9 El método *fold()* . . . 157
13.10 Herramientas de los Iteradores . . . 160

14 Funciones puras, inmutabilidad **163**
14.1 Distinguir entre datos, cálculos y acciones . . . 164
14.2 Tipos de entradas y salidas de las funciones . . . 169
14.3 Separar las cosas . . . 169
14.4 Extraer los cálculos de las acciones . . . 170
14.5 Ejemplo: eliminar entradas implícitas . . . 172
14.6 Inmutabilidad. Copy-on-write . . . 174

15 Colecciones **177**

16 Patrones de diseño en la Programación Funcional **179**

17 Grokking Simplicity **181**
17.1 Welcome to Grokking simplicity . . . 182
17.2 Functional thinking in action . . . 182
17.3 Distinguishing actions, calculations and data . . . 182
17.4 Distinción entre acciones, cálculos y datos . . . 185
17.5 Ejemplo . . . 187
17.6 Extracting calculations from actions . . . 188
17.7 Improving the design of actions . . . 188
17.8 Staying inmutable in a mutable language . . . 188
17.9 Staying inmutable with untrusted code . . . 188
17.10 Stratified design (I) . . . 188
17.11 Stratified design (II) . . . 188
17.12 First-class functions (I) . . . 188
17.13 First-class functions (II) . . . 188
17.14 Functional iteration . . . 188

17.15 Chaining functional tools . . . 188
17.16 Functional tools for nested data . . . 188
17.17 Isolating timelines . . . 188
17.18 Sharing resources between timelines . . . 188
17.19 Coordinating timelines . . . 188
17.20 Reactive and onion architectures . . . 188
17.21 The functional journey ahead . . . 188

18 Functional Programming made easier (Scalfani) 189
18.1 1.- Discipline is freedom . . . 189
18.2 Variables globales . . . 191

19 Railway Oriented Programming 197
19.1 Monoides . . . 197

20 NaN an Inf 199
20.1 Artículos para recomendar a los alumnos . . . 199

REFERENCIAS 201

Prólogo

La programación funcional no es nueva. El lenguaje LISP (LISt Processing) está considerado uno de los primeros lenguajes de alto nivel que se desarrollaron, junto a Fortran y Cobol. Durante mucho tiempo se ha considerado un paradigma de programación demasiado teórico y alejado de la resolución de los problemas habituales en la programación real.

Durante ese tiempo, la Programación Orientada a Objetos (POO) se consideraba el paradigma base necesario para la programación en cualquier lenguaje. Con el tiempo, la POO ha mostrado algunas de sus debilidades, originadas por el mecanismo de la herencia entre tipos. Ya en 1994 "*The gang of four*", en su famoso libro "*Design Patterns: Elements of Reusable Object-Oriented Software*", indicaban que había que primar la composición frente a la herencia al construir los programas [1].

Los principios que rigen la programación funcional son muy simples. En los últimos años la programación funcional ha adquirido especial relevancia. Todos los lenguajes principales han ido incorporando algunos de los elementos que caracterizan a la programación funcional, como los iteradores, las funciones de primera clase, las closures, la inmutabilidad de variables u otros. Es posible escribir programas con un estilo funcional casi en cualquier lenguaje.

Rust es un lenguaje de programación con solo unos años de vida pero que está teniendo un crecimiento muy importante debido la seguridad del manejo de memoria que proporciona, a la velocidad de ejecución del código generado, a la calidad de las herramientas que proporciona y a las buenas sensaciones que proporciona al programar. Rust no es un lenguaje funcional estricto, pero proporciona suficientes elementos para poder ser utilizado como tal.

A lo largo del curso, se va a hacer una introducción a la utilización del lenguaje Rust y a los conceptos que sustentan el paradigma de la programación funcional. Se utilizará principalmente el lenguaje Rust, aunque también se mostrarán ejemplos en otros lenguajes, como Java, C o Javascript.

Al final del curso, el alumno comprenderá los conceptos que subyacen bajo el paradigma de la programación funcional, de forma que podrá utilizar algunos de ellos en el lenguaje que utilice habitualmente. Ademas, conocerá los conceptos fundamentales de la programación con Rust, lo que le permitirá adentrarse en el lenguaje, si lo considera de interés.

Acerca del autor

Santiago Higuera de Frutos

Doctor Ingeniero de Caminos por la Universidad Politécnica de Madrid (UPM). Ha sido profesor en la Escuela de Ingenieros de Caminos y actualmente en Teleco Campus Sur. Es autor de los libros "*Programacion en Rust*" y "Programación y métodos numéricos para Ingeniería con MATLAB y Octave", publicados por la Editorial Garceta, así como de numerosos artículos y conferencias sobre programación, geometría de las carreteras y simuladores de conducción, todo ello en el ámbito del software de código abierto.

CAPÍTULO 1

Pensar de manera funcional. Programación declarativa

Contenido

1.1 Adiós a la Programación Orientada a Objetos
1.2 ¿En qué consiste la programación funcional?
1.3 Diferencias con la Programación Orientada a Objetos

Se considera que la programación funcional está basada en el cálculo lambda, un sistema formal desarrollado en los años 1930 para investigar la naturaleza de las funciones, de la computabilidad y su relación con la recursión.

El objetivo de este curso es explicar las técnicas y la forma funcional de razonar en programación para mejorar la calidad del código de los programas.

1.1 Adiós a la Programación Orientada a Objetos

A partir de los años 90 del siglo pasado, la Programación Orientada a Objetos (POO) se convirtió en el paradigma fundamental de programación. Si no sabías programar orientado a objetos, no sabías programar. Se enseñaba de una forma muy dogmática e incluso se suponía que había que *pensar orientado a objetos*. Cualquier programa se tenía que razonar en términos de clases y objetos, en lo que se denominaba *análisis y diseño orientado a objetos*.

A medida que se fueron desarrollando programas con esta metodología, se comprobó que la POO daba lugar a ciertos desarrollos que producían programas con código complicado, difícil de comprender y que además dificultaba la depuración y la realización de test. Se comprobó que la POO se adaptaba mejor a cierto tipo de problemas que a otros.

El tipo de datos en el que se basa la POO es la *clase*. A las instancias que se crean de una clase determinada se le llaman *objetos*. Las clases incluyen en una misma construcción de programación los datos, frecuentemente llamados *propiedades* y las funciones que operan sobre dichos datos, que se suelen llamar *métodos*. El valor de las propiedades representa el *estado* del objeto; los métodos de la clase representan el *comportamiento* del objeto y proporcionan la forma de modificar su estado.

La POO se basa en tres pilares fundamentales: *herencia*, *encapsulación* y *polimorfismo*.

En relación con el primero de estos pilares, la herencia, es cierto que funciona bien en las típicas jerarquías de clases sencillas que se ponen como ejemplo en los cursos básicos de programación. Es el caso de la clásica descomposición de clases que se muestra en la Figura 1.1.

El mundo real no siempre se puede modelizar bien haciendo una descomposición en categorías con propiedades bien definidas. Por ejemplo, se puede organizar una jerarquía para el reino animal que divida los animales en mamíferos, reptiles, aves, etc. Y cuando ya está organizada la jerarquía, aparece el ornitorrinco, que no encaja correctamente en ninguna de las categorías que se

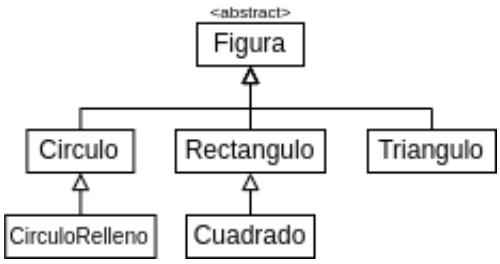

Figura 1.1: Descomposición típica de clases usada en los cursos de iniciación a la programación

habían previsto[1]. La solución puede ser crear una nueva categoría para el ornitorrinco o rehacer la jerarquía de clases para darle cabida. Cualquiera de las dos soluciones es muy costosa en términos de esfuerzo y complejidad.

Es frecuente que las jerarquías de clases sean muy profundas, con muchos niveles de clases que van heredando unas de otras. Las clases en lo alto de la jerarquía tienen métodos y propiedades que solo utilizan unas pocas clases, así como métodos para mantener estados que en muy rara ocasión se modifican.

A menudo, esta complejidad está originada por tratar de poner juntas cosas que nada tienen en común. Observe la Figura 1.2.

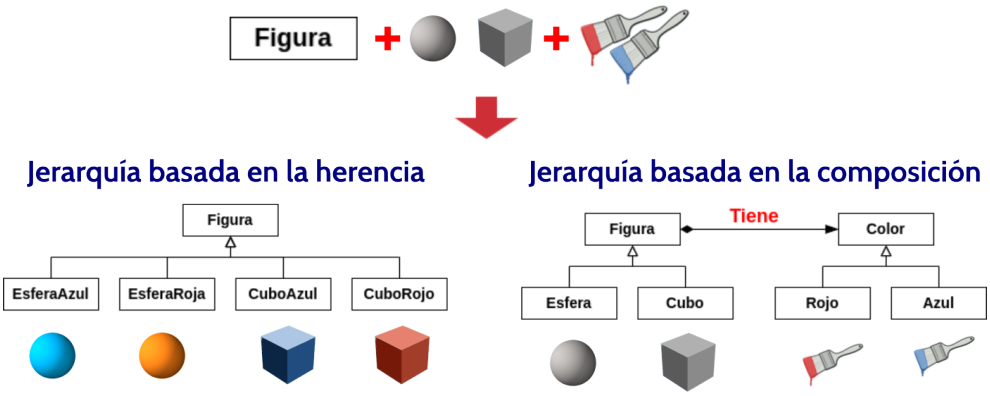

Figura 1.2: Diferentes jerarquías de clases según se priorice la herencia o la composición.

Si se están modelizando figuras, se tienen figuras como la esfera o el cubo. Pero si además se quiere dar cabida a esferas rojas o azules, cubos rojos o

[1] *"The platypus effect" (el efecto ornitorrinco)* es un término acuñado por Anselm Hook y del que yo he tenido conocimiento a través del artículo "*The Rise and Fall of Object Oriented Programming*" de David "Talin" Joiner.

azules, etc, se podría caer en crear categorías específicas para las esferas rojas, las esferas azules, los cubos rojos y los cubos azules. Erich Gamma et al., la *banda de los cuatro* (*the Gang Of Four, GoF*), ya indicaron en 1994 en su libro "Design Patterns" [1], que había favorecer la composición frente a la herencia. En la parte izquierda de la Figura 1.2 se puede ver la jerarquía de clases que se obtendría priorizando el mecanismo de herencia. En la parte derecha de la misma figura se puede ver el resultado cuando se aplica la composición.

Otro problema clásico asociado a la creación de jerarquías de clases es el llamado *problema del diamante*, que se esquematiza en la Figura 1.3. Los lenguajes no suelen permitir que una misma clase herede de dos clases antecesoras, por los problemas que pueden surgir para identificar que método concreto hay que aplicar en determinadas situaciones: ¿qué método *activar()* hereda la *Fotocopiadora*, el del *Scanner* o el de la *Impresora*? La solución, una vez más, es la composición: que la clase *Fotocopiadora tenga* una *Impresora* y un *Scanner*, no que derive de ellos. De esta forma, podrá decidir qué método *activar()* utilizará o, quizás, utilizar los dos, uno tras otro.

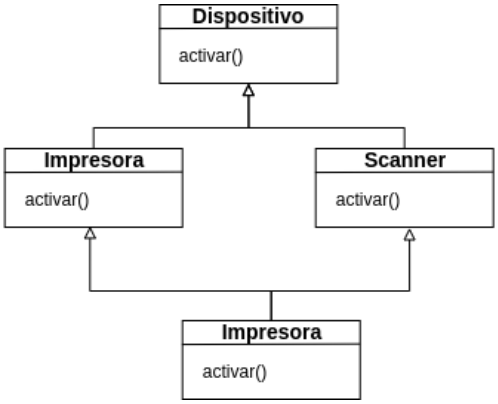

Figura 1.3: Problema del diamante en una jerarquía de clases

Un problema que no es menor en la POO es que, a medida que las jerarquías de clases crecen, se hace cada vez más difícil preparar test que permitan probar los nuevos métodos que se implementan. Cada nuevo método necesita mucho trabajo de preparación para los test, y esta complejidad se incrementa de manera exponencial a medida que crece la jerarquía de clases.

Un planteamiento alternativo es que las clases prácticamente solo tengan datos y que las funciones que se necesiten para operar sobre esos datos sean externas a las clases. Esta técnica da lugar a una organización del código mucho

más sencilla. Cada función solo realiza una tarea y cuando se quieren hacer test, solo hay que crear algunos juegos de datos de prueba y probar la función concreta, sin tener que luchar con una gran colección de clases y ficheros. Un caso típico donde este planteamiento es muy útil es cuando se trabaja con bases de datos relacionales, que se adaptan mal a la modelización de la POO.

En general, la composición da lugar a jerarquías de clases menos intrincadas, más sencillas de comprender y que se adaptan mejor a las pruebas, la depuración y las modificaciones.

Modelar el mundo a base de clases y objetos no deja de ser un tema subjetivo: cada programador puede encontrar diferentes formas de entender las categorías necesarias. De hecho, cada entidad de la vida real (clase) puede servir para cosas muy diferentes (métodos de la clase). Una taza puede servir para *beber()*, pero también podría servir como arma para *lanzar()* o como *pisapapeles()*.

Una de las *promesas* que se asociaba al paradigma de la POO es la reutilización. Suponga que tiene que desarrollar una aplicación y que ya dispone de una jerarquía de clases que se ha utilizado en una aplicación anterior. Surge la necesidad en la nueva aplicación de utilizar una clase similar a una que ya se utilizó en la aplicación anterior y se opta por copiar y pegar la clase en la nueva aplicación. Pero el programa no compila; dicha clase deriva de otra y esa otra de otra... Al final, es necesario trasladar a la nueva aplicación la clase que se necesita y toda la jerarquía de clases de las que hereda, que pueden no tener ninguna utilidad en la nueva aplicación. Pero sigue sin compilar; resulta que alguna de las clases de esa jerarquía utiliza otra clase que a su vez tiene su propia jerarquía. Al final, para reutilizar una clase concreta, hay que trasladar a la nueva aplicación un montón de clases que no se necesitan en absoluto... o hacer la clase nueva y olvidarse de reutilizar. Joe Armstrong, el creador del lenguaje Erlang, denominaba a este problema el *problema del gorila*: necesito un plátano y termino trayéndome el gorila que sostiene el plátano y, con él, toda su selva[2].

Hay problemas más sutiles asociados con la reutilización de clases. Observe el código Java siguiente. Se trata de una clase llamada *Array* que proporciona un método *add()*, para añadir elementos individuales y un método *addAll()*, que permite añadir un array ordinario de objetos en una sola operación. Podría tratarse del código de una librería y que solo pudiéramos utilizar los métodos que proporciona, sin que tuviéramos acceso al código fuente.

[2]Esta cita de Armstrong y algunas de las ideas y ejemplos que se muestran en este apartado proceden de artículos escritos por Charles Scalfani, autor del libro "*Functional Programming Made Easier*" [2].

```
class Array {
   private ArrayList<Object> a = new ArrayList<Object>();

   public void add(Object element) {
      a.add(element);
   }
   public void addAll(Object elements[]) {
      for (int i = 0; i < elements.length; ++i)
      a.add(elements[i]); // Esta línea va a cambiar
   }
}
```

Suponga ahora que, en nuestra aplicación, derivamos una clase llamada *ArrayCount*, que añade un contador a la clase *Array*. El código podría ser el siguiente:

```
public class ArrayCount extends Array {
   private int count = 0;

   @Override
   public void add(Object element) {
      super.add(element);
      ++count;
   }
   @Override
   public void addAll(Object elements[]) {
      super.addAll(elements);
      count += elements.length;
   }
   public int getCount() {
      return this.count;
   }
}
```

Nuestra clase sobreescribe los métodos de la clase base para gestionar el valor del contador. El código para probar nuestra clase *ArrayCount* podría ser el siguiente:

```
public class Main {

   public static void main(String[] args) {
      Integer[] list = {1, 2, 3};
      ArrayCount ac = new ArrayCount();
      ac.addAll(list);
      System.out.println(ac.getCount()); // Imprime 3
   }
}
```

Si se ejecuta, el programa mostrará que el contador de elementos vale 3, que es el valor correcto.

Ahora suponga que los creadores de la librería realizan un pequeño cambio en la clase base. El cambio solo afecta al método *addAll()*: en lugar de actuar directamente sobre el *ArrayList*, llaman a su propio método *add()*, como muestra el siguiente código:

```
public void addAll(Object elements[]) {
   for (int i = 0; i < elements.length; ++i) {
      add(elements[i]); // Esta línea ha cambiado
   }
}
```

Aparentemente el cambio es inocuo. Los creadores de la librería comprueban que la nueva función pasa todos los test y que el cambio no afecta en nada a la funcionalidad de la librería. Pero nuestro programa falla: si ejecutamos el programa, el contador muestra 6, en lugar de 3. No podemos saber qué está pasando, pues no tenemos acceso al código fuente y nos es imposible saber cuál es el problema.

Este problema podría llevarnos a la conclusión de que nunca hay que derivar una clase de otra de la que no tengamos acceso al código fuente.

Hay más problemas asociados a la creación de las jerarquías de clases. Las jerarquías de clases están pensadas para que las clases derivadas sean especializaciones de las clases antecesoras. El planteamiento es que las jerarquías se basen en *categorizar* las clases derivadas. Pero en el mundo real no se suelen encontrar jerarquías de ese tipo, es más frecuente encontrar jerarquías en

las que unas clases de objetos contienen a otros. En esos casos, la jerarquía de herencia que ofrecen los lenguajes de programación no se adaptan bien a la modelización de dichas jerarquías. Piense en el caso de una base de datos relacional, con sus tablas, sus registros, sus columnas, sus índices, sus relaciones entre las distintas tablas, etc. No está claro que clases derivan de otras, es más fácil pensar en términos de que unas clases *contienen* a otras. En esos casos, la herencia es de poca ayuda. Hay otros casos con problemas similares, por ejemplo la estructura de ficheros y directorios de un sistema de archivos, o muchas otras.

El segundo pilar en el que se apoya la POO es la encapsulación. Una de las reglas que se considera que debe cumplir cualquier diseño orientado a objetos es la *encapsulación*, según la cual, el estado de los objetos no se debe poder modificar directamente desde fuera del objeto, sino solo a través de los métodos que proporciona la clase en cuestión.

Por ejemplo, si se tuviera una clase denominada *Etiqueta* con una propiedad llamada *texto*, no se debería poder borrar el texto de la etiqueta mediante una instrucción del tipo:

```
etiqueta.text = “ ”;
```

Un diseño correcto proporcionaría un método *borrar()* que realizase dicha acción de borrar el texto de la etiqueta:

```
etiqueta.borrar();
```

Este procedimiento funciona perfectamente cuando las operaciones son sencillas, pero la cosa se complica cuando las acciones a realizar implican varios objetos de distintas clases. En esos casos, puede ser más sencillo utilizar funciones ajenas a las propias clases. Es una cuestión casi semántica: ¿en qué debemos poner más énfasis, en los sustantivos (*Etiqueta*) o en los verbos (borrar()).

La encapsulación parece proteger los datos internos de la clase de su acceso desde el exterior. Pero resulta que la mayoría de los lenguajes, cuando pasan un objeto a una función, lo que pasan no es una copia del objeto original, sino una referencia al mismo. Esto se hace por motivos de eficiencia.

Suponga el constructor de una clase recibe una referencia a un objeto que guarda en una propiedad privada interna. Por ejemplo, la clase *Circulo* recibe en su constructor un *Punto* que hace las veces de centro del círculo. Si lo que recibe *Circulo* es una referencia, el programa que está instanciando la clase tiene acceso a dicha referencia y podría modificar las coordenadas del punto sin utilizar los métodos de *Circulo*.

Una forma de resolver este problema es no pasar nunca referencias a objetos, pasar copias de los objetos. Hay que crear clones de los objetos realizados como *deep copies*, esto es, hay que copiar recursivamente todos los niveles del objeto que se quiere clonar. En el caso del *Circulo*, en vez de pasar un *Punto* en el constructor, podríamos pasar directamente las coordenadas *x* e *y* del centro. En todos los lenguajes, los tipos primitivos como los enteros o los *doubles*, cuando se pasan a una función, lo que se pasa es una copia del valor, no una referencia al mismo.

Pero no todos los objetos se pueden copiar. En los programas se usan referencias a componentes del sistema operativo como los manejadores de ficheros que no se pueden clonar, o *sockets* de una conexión u otros elementos que no admiten ser clonados. En esos casos, la propiedad interna del objeto estaría siempre expuesta a su acceso desde el exterior, rompiendo la regla de la encapsulación.

El tercer pilar en el que se basa la POO es el polimorfismo. No es que el polimorfismo no sea bueno, es que para disponer de él no es necesario un lenguaje orientado a objetos, es suficiente con implementar el polimorfismo basado en la herencia de interfaces.

Esta última, es la solución que ha implantado Rust. En Rust, los objetos se modelizan con los tipos *struct* o *enum*. Como se verá en el Apartado 3.4, los tipos *struct* y *enum* tienen datos y métodos, pero no disponen de un mecanismo de herencia entre tipos. En Rust, el tipo *trait* sería el equivalente a los interfaces de otros lenguajes. Los *traits* sí tienen la posibilidad de implementar el mecanismo de herencia. También se puede imponer que cualquier tipo de datos *implemente* uno o más *traits*. Con ello y con los denominados *tipos genericos*, se dispone de todos los elementos necesarios para utilizar un polimorfismo más flexible y con menos acoplamientos que el que resulta del polimorfismo basado en la herencia entre objetos. En el Apartado 3.15 se explicará cómo hacerlo.

En Java también es posible utilizar los interfaces y la herencia entre interfaces para conseguir este tipo de polimorfismo. De hecho, priorizando este mecanismo y la composición en vez de la herencia, se consiguen códigos más flexibles y desacoplados que siguiendo el método tradicional jerarquías basadas en la herencia.

1.2 ¿En qué consiste la programación funcional?

Es difícil definir el concepto de *Programación Funcional* (PF). Se considera que la PF está incluida en el paradigma de la *programación declarativa*, en la cual los

lenguajes se centran en describir qué quieren hacer en lugar de detallar cómo hacerlo, como sucede en los lenguajes imperativos.

Los siguientes elementos formarían parte del conjunto de componentes que utilizan los lenguajes funcionales:

- **Funciones puras:** se dice que una función es *pura* si no produce efectos secundarios, no depende de ningún estado exterior y siempre proporciona el mismo resultado para el mismo valor de los argumentos de entrada. Se entiende por *efecto secundario* cualquier acción de la función que afecte a algo fuera del contexto de la propia función. Por ejemplo, modificar el estado de una variable global, mostrar un mensaje en una pantalla o enviar un correo electrónico serían ejemplos de efectos secundarios.
- **Funciones de orden superior:** las funciones que operan utilizando otras funciones se denominan *funciones de orden superior*. En la programación funcional, las funciones se consideran *elementos de primera clase*, lo que significa que se pueden tratar como cualquier otro tipo de datos: se pueden utilizar como parámetros o como valor devuelto por otras funciones y pueden asignarse a variables. En la mayoría de lenguajes hay elementos que no son de primera clase, por ejemplo los operadores o las cláusulas que definen los bucles o las bifurcaciones.
- **Closures:** en programación funcional es habitual utilizar un tipo especial de funciones denominadas *closures*. Se trata de funciones anónimas definidas in situ que son capaces de capturar el entorno en el que fueron definidas, de forma que pueden acceder con posterioridad a variables de ese entorno con los valores que tuvieran en el momento en el que se declaró la *closure*. Actualmente, las *closures* también se incluyen en numerosos lenguajes, a veces bajo la denominación de *funciones anónimas*.
- **Inmutabilidad:** la programación funcional pone especial énfasis en la utilización de estructuras *inmutables*. Cambiar el valor de una variable, cambiar su estado, se considera un efecto secundario que hay que tratar de evitar. Decir que las *variables* no deben variar puede parecer un contrasentido, pero no es excepcional en los lenguajes de programación. Por ejemplo, en Java o en Javascript, las variables del tipo *String* son inmutables. Cuando se quiere modificar el estado de una variable inmutable lo que se hace es crear una nueva variable con el nuevo valor. El procedimiento consiste en *transformar* una variable en otra, en vez de en modificar el estado de la variable original. La inmutabilidad facilita los test de los programas y hace más segura la utilización de la programación multiproceso o en paralelo.

- **Recursividad:** el control de flujo en la programación funcional favorece la recursividad frente a los bucles. Sustituyendo los bucles del tipo *for* o *while* mediante recursividad se consigue un código más declarativo y, en ocasiones, más elegante.
- **Iteradores:** los *iteradores* son una construcción habitual en la programación funcional y que, a día de hoy, incorporan muchos lenguajes. Son estructuras que permiten recorrer una colección de manera ordenada, sin recurrir a bucles y variables de índice.
- **Evaluación perezosa:** las expresiones, siempre que se pueda, se deben comportar de manera *perezosa*. La *evaluación perezosa* (*lazy evaluation*) consiste en que determinados cálculos que impliquen a variables no se realicen hasta que son estrictamente necesarios.
- **Sin estado ni efectos secundarios:** en la programación funcional se trata de minimizar los estados mutables y los efectos secundarios. El resultado es un código en el que es más fácil razonar, hacer test y realizar la depuración de errores. Cuando es necesario mantener estados o realizar acciones con efectos secundarios, se controla rigurosamente.

Tras leer los párrafos anteriores, el lector puede estar preguntándose cómo es posible realizar un programa de aplicación práctica sin efectos secundarios y sin cambiar el valor de las variables. Bien, no es posible, los programadores funcionales utilizan funciones impuras en numerosas ocasiones y necesitan utilizar variables mutables en determinados contextos. No obstante, durante el desarrollo de un programa, hay numerosas situaciones en las que la utilización de funciones puras y el respeto a la inmutabilidad proporciona más seguridad y da lugar a que el código generado sea más escalable.

La programación funcional no es una sintaxis determinada, consiste en una serie de técnicas orientadas a eliminar los efectos secundarios o, al menos, limitar su alcance. Si se utilizan estas técnicas de manera adecuada, se consigue escribir código más fácil de leer, más correcto, más seguro, más fácil de probar y más fácil de depurar, lo que a fin de cuentas es el objetivo fundamental que se debe perseguir al programar.

La técnicas de la programación funcional no están restringidas por el lenguaje de programación que se utilice. Los conceptos que se utilizan en la programación funcional se pueden aplicar a la programación orientada a objetos o a la programación basada en procedimientos. Son principios generales que producen beneficios en la codificación de cualquier programa y en cualquier lenguaje. En realidad se trata de buenas prácticas de codificación de carácter universal.

1.3 Diferencias con la Programación Orientada a Objetos

Mientras que la Programación Orientada a Objetos se centra en la interacción y la comunicación entre diferentes objetos, la Programación Funcional se centra en cómo se van transformando los objetos. Dicha transformación hay que entenderla en el sentido de que un objeto se le pasa como argumento a una función que devuelve un nuevo objeto que incorpora las transformación que se quiere realizar. El objeto que devuelve la función es un nuevo objeto y el objeto original se puede conservar o desechar. En la Programación Funcional, unos objetos se transforman en otros, no se modifican.

Se resumen a continuación algunas de las características de la programación orientada a objetos (POO):

- **Estado de los objetos:** la POO está basada en objetos que encapsulan en una sola entidad su estado (propiedades) y su comportamiento (métodos). La *mutabilidad* es una parte inherente de la POO. El estado de los objetos puede variar a lo largo del tiempo, lo que se consigue a través de los métodos que proporcionan las clases. De esta manera, se modelizan las entidades reales y sus relaciones. Esta mutabilidad es una forma natural de representar las propiedades de los objetos que necesitan cambiar de valor a lo largo del tiempo, pero da lugar a problemas complejos para gestionar estados globales y secundarios, por ejemplo en la programación concurrente.
- **Clases y herencia entre tipos:** la POO se basa en el concepto de clases como modelos de los objetos y a menudo utiliza la herencia entre clases para compartir y ampliar el comportamiento de las mismas.
- **Polimorfismo:** objetos de diferentes clases pueden ser tratados como si fueran de una misma clase base de la cual todos derivan.
- **Paradigma imperativo:** en general, la POO utiliza código con un paradigma imperativo, describiendo los pasos que hay que dar para modificar el estado de los objetos.

En el caso de la programación funcional (PF), algunas características distintivas son:

- **Funciones sin estado:** está basada en la utilización de funciones que no mantienen ningún estado y operan sobre datos inmutables. En ese sentido, la PF separa el estado de los objetos de su comportamiento. El estado se representa mediante estructuras de datos inmutables. El comportamiento se expresa a través de funciones que operan sobre dichos datos.

- **Funciones como objetos de primera clase:** las funciones son objetos de primera clase que se pueden utilizar como parámetros de otras funciones, se pueden devolver como resultados y se pueden asignar a variables. Las funciones se utilizan para modelizar abstracciones, encapsular comportamientos. Para conseguir procesamientos complejos se utiliza la composición de funciones.
- **Paradigma declarativo:** la PF utiliza una codificación más declarativa, expresando la lógica de lo qué se quiere hacer, sin describir el flujo de instrucciones para ello.

La opinión del autor es que no hay que ser fundamentalista de ningún paradigma de programación. Hay que recordar el viejo dicho: "*al que la única herramienta que conoce es el martillo, todo le parecen clavos*". Hay problemas que se adaptan mejor a unas técnicas de programación u otras. En muchas ocasiones, una combinación adecuada de las técnicas de la POO y de la PF será lo más adecuado. En todos los casos hay que analizar el problema que se trata de resolver y utilizar la combinación de técnicas que mejor se adapte al mismo.

CAPÍTULO 2

“¡Hola, Mundo!” en Rust

Contenido

2.1 Instalación de Rust
2.2 ¡Hola, mundo! en Rust
2.3 ¡Hola, mundo!, utilizando Cargo
2.4 Código de los ejemplos
2.5 Conceptos comunes en programación
2.6 Comentarios en los programas

Para poder seguir el contenido del curso es importante tener instalado el entorno de desarrollo del lenguaje Rust. La instalación es sencilla y rápida, si se siguen las instrucciones que se dan en la propia página del proyecto y que se indicarán en este primer capítulo de la documentación del curso.

Además, es importante utilizar un editor de texto con los complementos necesarios para trabajar con Rust. Los contenidos del curso se han desarrollado utilizando Visual Studio Code de Micrososft y RustRover de Jet Brians. Ambos son editores de gran calidad y que ofrecen importantes ayudas a la programación en Rust.

Esté capítulo de la documentación del curso se completa con la explicación de cómo desarrollar el clásico programa "¡Hola, Mundo!" y con la explicación de algunos conceptos comunes, como la inclusión de comentarios en el código, la utilización de algunas macros del lenguaje útiles en entornos de depuración y pruebas y la posible utilización del portal "Rust Playground".

2.1 Instalación de Rust

Para instalar el entorno de desarrollo de Rust, el método más sencillo es descargarse el *script* de instalación que se proporciona para cada sistema operativo y ejecutarlo en el ordenador. Las instrucciones se dan en la página Web de Rust, en la siguiente dirección:

```
https://www.rust-lang.org/tools/install
```

Los usuarios de Linux y de Mac no tendrán problema en seguir las instrucciones que se dan en dicha página. Los usuarios de Windows pueden descargar el instalador directamente del siguiente enlace:

```
https://static.rust-lang.org/rustup/dist/i686-pc-windows-gnu/rustup-init.exe
```

Para poder compilar y construir los programas se necesita un *linker*. Es posible que ya lo tenga instalado en su sistema. Si aparecieran errores relativos a la falta del *linker*, hay que instalar un compilador de `C`, que suele tener el *linker* ya incorporado. En cualquier caso, es útil tener un compilador de `C` instalado en el sistema, pues algunos paquetes pueden depender de él para su correcto funcionamiento.

Los usuarios de *Linux* pueden instalar `GCC` o `Clang`, el que sea acorde a su distribución. Los usuarios de *Ubuntu* pueden instalar el paquete `build-essentials`.

Los usuarios de *macOS* pueden obtener un compilador de `C` tecleando la siguiente instrucción:

```
xcode-select -install
```

Los usuarios de *Windows* deben seguir las instrucciones de instalación que se dan en la página web del proyecto Rust [3], en la siguiente dirección web:

https://www.rust-lang.org/tools/install

Durante la instalación, se le indicará al usuario que necesita tener instaladas las herramientas de C++ para *Visual Studio 2013* o posterior. Se pueden instalar las herramientas para *Visual Studio 2019* comprobando que se activa la opción correspondiente a la instalación de las *C++ build tools*, el *SDK* para *Windows 10* y los componentes en inglés de dicho *pack*.

La instalación que hace `rustup` incluye el programa `rustup`, el compilador `rustc` y el gestor de paquetes *Cargo*. Para comprobar que el conjunto de desarrollo está correctamente instalado en el ordenador, se pueden probar las siguientes ordenes en la consola:

```
rustup show
rustc --version
cargo --version
```

El resultado debería ser similar al de la Figura 2.1.

```
>> rustup show
Default host: x86_64-unknown-linux-gnu
rustup home:  /home/shiguera/.rustup

stable-x86_64-unknown-linux-gnu (default)
rustc 1.57.0 (f1edd0429 2021-11-29)
>>
>> rustc --version
rustc 1.57.0 (f1edd0429 2021-11-29)
>>
>> cargo --version
cargo 1.57.0 (b2e52d7ca 2021-10-21)
>>
```

Figura 2.1: Instrucciones en el terminal para comprobar las versiones instaladas de los programas *rustup*, *rustc* y *cargo*

Si no aparecen los mensajes y se está trabajando en *Windows*, debería comprobar que Rust está en la variable `%PATH%` del sistema. En adelante, las instrucciones que se indican en este libro deberían de funcionar en los terminales de cualquiera de los tres sistemas operativos.

Para actualizar a una versión más reciente de Rust, hay que teclear en el terminal la siguiente instrucción:

```
rustup update
```

Para desinstalar Rust hay que teclear:

```
rustup self uninstall
```

Al instalar Rust, se carga en el ordenador una versión de la documentación. Mediante la siguiente instrucción, se puede consultar dicha documentación localmente en el navegador:

```
rustup doc
```

Podrá navegar a través de multitud de documentos y libros que le servirán para profundizar sobre Rust. En particular, un buen comienzo es el "Libro de Rust", escrito y mantenido por Steve Klabnik y Carol Nichols, con contribuciones de la *Rust Community*. Además de la copia que se instala en su ordenador, puede consultar la versión más actualizada en la web, en la referencia [3]:

```
https://doc.rust-lang.org/book/
```

El portal oficial de Rust ofrece enlaces a diversos documentos de interés en la siguiente dirección:

```
https://www.rust-lang.org/learn
```

Cuando le surjan dudas acerca de alguna función o instrucción de las que se explican en este libro, la mejor manera de acceder a una documentación más completa de la misma es la referencia de la API de la librería estándar. La documentación local le ofrecerá una copia, o también la puede consultar en la web en la referencia [4].

A día de hoy, existen numerosos libros donde iniciarse o profundizar en los distintos aspectos del lenguaje. Como no podía ser de otra manera, el autor de estas líneas considera que su libro "Programación en Rust", publicado en la Editorial Garceta, es una buena herramienta para aprender a programar en Rust [5].

Otras editoriales que disponen de libros sobre Rust son:

- *Packt*: `https://www.packtpub.com/`
- *Manning*: `https://www.manning.com/`
- *O'Reilly*: `https://www.oreilly.com/`

2.1.1 El editor de texto

Para completar el entorno de trabajo, será necesario disponer de un editor. El código de los programas en lenguaje Rust se escribe en ficheros de texto plano. Se

puede utilizar cualquier editor de texto para ello. En el siguiente enlace se muestran los editores que están comprobados por la fundación Rust y que proporcionan herramientas de compilación integradas, ayudas y corrección de sintaxis y otras:

`https://www.rust-lang.org/tools`

El autor utiliza habitualmente Visual Studio Code de Microsoft y RustRover de JetBrains.

Visual Studio Code de *Microsoft* [6], es una buena opción. Hay versiones para cualquier sistema operativo, es gratuito, de buena calidad y dispone de una buena integración con Rust. El nombre con el que se suele denominar al programa es *VS Code*.

Para sacar el máximo provecho de la utilización de *VS Code* al programar en Rust, es conveniente instalar la extensión existente para Rust, que ofrece numerosas ayudas, como autocompletado, remarcado de errores, terminal para ejecutar las órdenes de compilación[1], gestión de paquetes y otras utilidades que facilitan mucho la gestión de los programas en Rust. Se puede consultar cómo hacerlo en:

`https://code.visualstudio.com/docs/languages/rust`

RustRover está desarrollado por JetBrains. JetBrains dispone de editores de gran calidad para varios lenguajes. Son de pago, pero ofrece versiones completas de manera gratuita para estudiantes y profesores. Utilizando una cuenta de correo de la UPM, se puede disfrutar de dichas ventajas en todos los editores de JetBrains. El editor RustRover se puede desacargar desde la siguiente dirección:

`https://www.jetbrains.com/rust/`

Los dos editores comentados son de gran calidad y también lo serán, con seguridad, el resto de editores recomendados por la Fundación Rust, aunque el autor no ha tenido ocasión de probarlos.

[1] En la opción del menú `View::Terminal`, el programa *VS Code* ofrece un terminal para ejecutar todos los comandos. En el caso de los usuarios de *Windows*, el terminal ofrecido es *Power Shell*, aunque se puede configurar para utilizar el terminal `cmd` estándar.

2.2 ¡Hola, mundo! en Rust

Siguiendo la tradición, se va a explicar cómo crear el primer programa en Rust, un programa que imprima `¡Hola, mundo!` en el terminal del ordenador. Se supone que se está trabajando en el terminal del sistema. Se recomienda crear un directorio para los programas escritos en Rust y, dentro de él, directorios individuales para cada programa. Para este primer programa, la forma de proceder en *Linux, macOS* o *Windows* con *Power Shell* podría ser la siguiente:

```
mkdir ~/rust_projets
cd ~/rust_projects
mkdir hola_mundo
cd hola_mundo
```

En el caso de estar trabajando en la terminal `cmd` de *Windows*, las instrucciones equivalentes serían las siguientes:

```
mkdir "%USERPROFILE%\rust_projects"
cd /d "%USERPROFILE%\rust_projects"
mkdir hola_mundo
cd hola_mundo
```

Con ello, se ha creado un directorio llamado `rust_projects` en el directorio del usuario y, dentro de él, un directorio llamado `hola_mundo`. Tras crear los directorios, nos hemos posicionado dentro del directorio `hola_mundo`. Obsérvese que, para el nombre de los directorios del proyecto, se ha utilizado lo que será el convenio habitual para nombrar programas, variables y funciones en el lenguaje Rust, el denominado *snake case*: letras minúsculas separando las palabras con el guión bajo.

A continuación, utilizando el editor de texto, cree un fichero llamado `main.rs` para teclear el código del programa. Los ficheros de código en Rust tienen la extensión `rs`. Se podría llamar al fichero con otro nombre, pero también es convenio que el programa que contiene la función `main()` se llame `main.rs`. Esta función es lo primero que se ejecuta en los programas escritos en Rust. Teclee el siguiente código en la función recién creada:

```
fn main() {
   println!("¡Hola, mundo!");
}
```

La primera línea es la signatura[2] de la función *main()*. Como se puede observar, consta de la palabra clave *fn*, que indica al compilador que se está declarando una función, seguida por el nombre de la función, en este caso *main*. A continuación se escriben dos paréntesis, en esta ocasión vacíos. Dentro de estos paréntesis se escriben los parámetros que admite la función. En este caso, la función *main()* no recibe ningún parámetro. A continuación, se pondría el tipo de datos del valor devuelto por la función, pero en este caso la función *main()* tampoco devuelve ningún valor.

A continuación se escribe el código de la función, encerrado entre una pareja de llaves `{}`. El convenio de escritura en Rust es escribir la llave de apertura en la línea de signatura; las instrucciones que componen el cuerpo de la función se escriben en las lineas siguientes, tabuladas hacia la derecha; finalmente, la llave de cierre de la función se escribe en una línea nueva tras la última instrucción, alineada con la palabra `fn` de la signatura.

Al instalar Rust, se instala también la utilidad `rustfmt` que permite formatear los ficheros de código fuente siguiendo los convenios de escritura de Rust. Para ello, solo hay que teclear en el terminal la orden siguiente:

```
rustfmt nombre_del_fichero_fuente
```

El código de la función `main()` recién creada consta de una sola línea. Se trata de la macro `println!()` que escribe una línea de texto en la pantalla. En este caso se le ordena escribir la cadena de caracteres `"¡Hola, mundo!"`. Se hablará algo más de la macro `println!()` en el Apartado 2.4.

Para poder ejecutar el programa hay que compilarlo. Esto se consigue tecleando la siguiente instrucción en el terminal:

```
rustc main.rs
```

El programa `rustc` es el compilador de Rust. Al ejecutar la instrucción anterior, se compila el programa y se crea un ejecutable, llamado `main`.

[2] En teoría de programación, se denomina *signatura* de una función a la primera línea, en la que se declara el nombre, los parámetros y el valor devuelto por la función.

Si tras esa instrucción se listan los ficheros del directorio, se ve que hay un fichero ejecutable `main`. En *Windows*, el ejecutable se llama `main.exe` y además, se crea otro fichero llamado `main.pdb` que contiene instrucciones internas de compilación.

Para ejecutar el programa, en *Linux* y *macOS* hay que teclear la siguiente instrucción:

```
./main
```

En el terminal `cmd` de *Windows*, la instrucción sería:

```
.\main.exe
```

En ambos casos, se obtendrá en pantalla la frase:

```
¡Hola, mundo!
```

2.3 ¡Hola, mundo!, utilizando Cargo

El proceso descrito en el apartado anterior, consistente en compilar directamente con `rustc` el fichero fuente, puede servir para programas que constan de un solo fichero fuente. Pero, en general, los programas constarán de más de un fichero fuente y es conveniente organizar el código de todos los ficheros y el proceso de compilación de manera eficiente.

Una de las utilidades que queda instalada en el ordenador con Rust es el gestor de paquetes *Cargo*. Se trata de un gestor de paquetes muy completo que facilita mucho la tarea de creación y mantenimiento de los programas escritos en lenguaje Rust.

El gestor *Cargo* se encarga de construir y compilar los programas. También se encarga de gestionar las *dependencias* de los proyectos. En Rust, se denominan *dependencias* las librerías externas que se deban incorporar al programa. Una vez definidas las dependencias del proyecto, *Cargo* se encarga de descargarlas, mantenerlas actualizadas y compilarlas. Existen miles de librerías disponibles para Rust. Puede consultarlas en el repositorio público “The Rust community’s crate registry” [7].

Se va a realizar ahora un programa tipo “*Hola Mundo*”, similar al realizado en el apartado anterior, pero utilizando el gestor de paquetes *Cargo*. Para ello, desde el terminal, sitúese en el directorio que creó para albergar los proyectos de Rust y teclee la siguiente instrucción:

```
cargo new hola_cargo
```

Con esta instrucción, el programa *Cargo* creará una nueva carpeta de nombre `hola_cargo` y, dentro de ella, varias carpetas y archivos que constituyen un programa ejecutable. Si se sitúa dentro de la carpeta del proyecto recién creado y lista los archivos y carpetas creados por *Cargo* al crear el programa, obtendrá un resultado similar al de la Figura 2.2.

```
>> ls -A
Cargo.toml  .git  .gitignore  src
>>
```

Figura 2.2: Ficheros y carpetas creados en el directorio del proyecto *hola_cargo*, tras su creación por *Cargo*

En la Figura 2.2 se puede ver que se ha creado el fichero `Cargo.toml` [3]. Este fichero contiene la información necesaria para compilar el programa. También se incluirían ahí las dependencias, si las hubiera. El contenido de dicho fichero, en el proyecto recién creado es similar al siguiente:

```
[package]
name = "hola_cargo"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Las secciones de este fichero comienzan con un nombre de sección entre corchetes. La primera sección, `[package]`, contiene el nombre y la versión del programa y la edición de Rust utilizada para su creación. A continuación hay unos comentarios, que son las líneas que comienzan con el carácter #. Finalmente, hay una sección `[dependencies]` inicialmente vacía.

Además del fichero `Cargo.toml`, en la carpeta del proyecto recién creado aparece un fichero `.gitignore` y una carpeta `.git`. Al crear un nuevo programa, *Cargo* inicializa un repositorio *GIT* para el mismo. Se pueden utilizar también otros gestores de versiones.

[3] TOML es el acrónimo de «*Tom's Obvious, Minimal Language*». Este lenguaje está pensado para los ficheros de configuración. Su creador fue *Tom Preston-Werner*.

También se puede crear un nuevo proyecto Rust en un directorio donde ya exista un repositorio *GIT*, mediante la orden `cargo init`. Puede teclear en la consola la orden `cargo new -help` para obtener más información o consultar el «Libro de Cargo» [8] en la documentación de Rust.

En el directorio del proyecto hay un directorio llamado `src` y, dentro de él, un fichero llamado `main.rs`. Si edita el fichero `main.rs` verá que contiene un código como el siguiente:

```
fn main() {
   println!("Hello, world!");
}
```

Es la función `main()` con la misma instrucción `println!()` que se utilizó en el apartado anterior. Cada vez que *Cargo* crea un nuevo programa, lo crea como un programa «*Hola Mundo*» listo para funcionar.

Para compilar y ejecutar el programa hay que teclear en el terminal la siguiente instrucción:

```
cargo run
```

Si vuelve a listar el directorio del proyecto verá que se ha creado una nueva carpeta llamada `target`. Dentro de ella hay otras carpetas y ficheros. El ejecutable creado está en la carpeta `target/debug` y lo podrá ejecutar en *Linux* y *macOS* con la instrucción siguiente:

```
./target/debug/hola_cargo
```

En *Windows*, la instrucción que hay que teclear para ejecutar el programa es:

```
.\target\debug\hola_cargo.exe
```

En el directorio del proyecto verá que también se ha creado un fichero `Cargo.lock`. Se trata de un fichero que utiliza el programa *Cargo* para gestionar las versiones utilizadas de las dependencias. Dicho fichero lo mantiene de manera automática el programa *Cargo* y el usuario no necesita modificarlo. Si tiene curiosidad, puede listar el fichero para ver su contenido.

Se puede compilar o construir el programa sin ejecutarlo, mediante la siguiente instrucción:

```
cargo build
```

El comando `cargo build` admite diferentes opciones de compilación en com-

binación con las especificaciones del fichero `Cargo.toml`. Tienen que ver con la versión a construir, la realización de test y otras. Se recomienda consultar el "*Libro de Cargo*" [8] para ampliar la información al respecto.

A veces, la primera vez que se construye el programa o tras varias sesiones de trabajo consecutivas, puede suceder que el comando `cargo build` no funcione de la manera esperada. Una posible solución es ejecutar el comando `cargo clean`, que eliminará el directorio `target`. Tras esta limpieza, el comando `cargo build` debería volver a funcionar de manera adecuada.

Otra opción interesante es `cargo check`, que comprueba el código sin llegar a compilar el ejecutable. En programas extensos puede ser más rápido que compilar el programa completo.

En Rust, los programas se denominan "*crates*". Hay dos tipos de *crates*: programas binarios (ejecutables) y librerías. Los programas que hemos visto hasta ahora entran en la categoría de programas ejecutables. Para crear una librería, la orden que hay que teclear en el terminal es la siguiente:

```
cargo new --lib mi_libreria
```

El programa *Cargo* creará la estructura de directorios correspondiente. Dentro del directorio *src* crea un fichero *lib.rs* con el código inicial de la librería creada, que consiste en un test.

2.4 Código de los ejemplos

A lo largo de este texto se dan numerosos ejemplos de código. Sería muy farragoso crear un nuevo proyecto para cada ejemplo. Una manera más eficiente de probar el código de los ejemplos puede ser utilizar *Cargo* para crear un proyecto genérico de nombre arbitrario y luego limitarse a cambiar el código de la función *main()* por el código que aparece para cada ejemplo concreto. Tras ello, se puede comprobar el resultado ejecutando la orden *cargo run*.

Otra forma de probar los ejemplos es el portal *Rust Playground*, que permite probar los programas en línea y ofrece algunas opciones adicionales interesantes:

```
https://play.rust-lang.org/
```

En el caso de los ejemplos escritos en Java, hay diversos portales online donde probar código Java. Una buena opción la puede encontrar en el siguiente enlace:

```
https://onlinegdb.com/2ea9z_GhF
```

En algunos ejemplos, se incluye código que da lugar a errores de compilación, que se indicarán en los comentarios del código. Se hace a propósito, con fines didácticos y para que el lector pueda observar los mensajes de error que muestra el compilador de *Rust*. Una de las características destacadas de *Rust* es la calidad de los mensajes de error y las sugerencias para resolver dichos errores que ofrece el compilador. La lectura detenida de estos mensajes constituye también una valiosa fuente de aprendizaje.

A lo largo de los ejemplos se utilizan profusamente la instrucción `println!()` y la instrucción `assert_eq!()`. Se trata de macros del lenguaje. Las macros son un tipo especial de funciones; se diferencian de las funciones ordinarias en que el último carácter del nombre es el símbolo de exclamación "`!`". El uso básico de estas dos macros en los ejemplos se hace necesario. Se supone que el lector comprenderá rápidamente su funcionamiento cuando ejecute por sí mismo los ejemplos que se proponen. Se da a continuación una pequeña explicación de su funcionamiento:

- **`println!()`:** esta macro permite hacer salidas por pantalla. Recibe una cadena de caracteres. Dentro de la cadena se pueden incluir *especificaciones de formato*, que son una pareja de llaves " `{}`". En la salida, la pareja de llaves se sustituye por el valor de las variables que aparecen listadas tras la cadena de caracteres. Por ejemplo:

```
println!("La variable x vale {}", x);
```

  Esta instrucción escribirá en pantalla la cadena `"La variable x vale "`, seguida del valor que tenga en ese momento la variable `x`.
- **`assert_eq!()`:** esta macro compara los valores de los resultados de las dos expresiones que recibe como parámetros. Si son iguales, no hace nada, pero si no lo son, interrumpe la ejecución del programa con un mensaje de error.

2.5 Conceptos comunes en programación

En la década de los años treinta del siglo XX, Alan Turing estableció los principios que debía cumplir un sistema de programación para poder resolver cualquier problema de computación. Traducido a los conceptos que se manejan en los lenguajes de programación actuales, se pueden resumir en tres condiciones:

- **Asignación de variables:** las variables son nombres simbólicos (etiquetas) asociados a valores en memoria. Mediante el procedimiento de asig-

nación, se asocia el nombre simbólico a un valor guardado en la memoria. De esta forma, una vez que se declara y se asigna valor a una variable, queda guardado en memoria y dicho valor se podrá utilizar en otra parte del programa, recuperándolo mediante el nombre de la variable

- **Bifurcaciones:** procedimiento que permite variar el flujo del programa, en base a condiciones lógicas
- **Bucles:** procedimiento que permite repetir cierto número de veces un bloque de código

Además de estos conceptos, casi todos los lenguajes permiten utilizar determinados tipos básicos de datos, permiten utilizar líneas de comentarios en el código fuente del programa y proporcionan la posibilidad de definir funciones.

En *Rust* es importante distinguir entre las instrucciones que dan lugar a un valor y las que no. Se denominan *declaraciones*, (*statements*), las instrucciones que realizan alguna acción pero sin devolver ningún valor. Por ejemplo, la siguiente instrucción es una declaración.

```
let x=7;
```

Se denominan *expresiones*, (*expressions*), las instrucciones que dan lugar a un resultado. Por ejemplo, la siguiente instrucción es una expresión:

```
x+y
```

El concepto existe en todos los lenguajes, pero en algunos no es muy importante. En cambio, en *Rust* es una diferencia que adquiere importancia en numerosas situaciones, como se irá viendo a lo largo de esta documentación.

Todos los conceptos de programación comentados en los párrafos precedentes son comunes a casi todos los lenguajes de programación, si bien cada uno los resuelve con diferente sintaxis o mediante procedimientos específicos.

Otra particularidad común a todos los lenguajes es la existencia de palabras reservadas del lenguaje, que no se pueden utilizar como nombres de variables o funciones. Una vez más, cada lenguaje tiene su propio conjunto de palabras reservadas.

2.6 Comentarios en los programas

El código de los programas puede incluir comentarios informativos. Se puede hacer de dos maneras: comentarios de una línea o comentarios de bloque de varias líneas.

Los comentarios de línea, se añaden anteponiendo dos barras inclinadas `//` al texto que se quiera usar como comentario. Si las barras están al principio de la línea, toda la línea será un comentario. Si están tras alguna instrucción, el comentario será desde donde aparecen las barras inclinadas hasta el final de la línea.

También se pueden hacer comentarios de bloque, que abarquen varias líneas. Se hacen poniendo el bloque comentario entre los símbolos `/*` y `*/`. Todo lo que quede entre los dos símbolos, se tratará como un comentario:

```
// Comentario de una línea
let x = 2; // Comentario de parte de una línea
/* Este es un comentario de bloque,
que abarca varias líneas de texto.
El criterio de estilo es poner los símbolos
de apertura al principio de la primera línea,
y los símbolos de cierre, al principio de la
siguiente línea a la última de texto comentado
*/
```

En general, el criterio de estilo al codificar programas en *Rust* es utilizar comentarios con doble barra inclinada, incluso si ocupan más de una línea. En los ejemplos que acompañan a las explicaciones de los capítulos, se verá en numerosas ocasiones la utilización de comentarios.

A veces, los comentarios se utilizan para inutilizar algunas líneas de código y que no se ejecuten. Es una técnica habitual durante la depuración de los programas.

Hay otro tipo de comentarios que sirven para generar la documentación de los programas, pero que no se tratan en este curso. El lector interesado puede consultar la forma de documentar programas en el siguiente enlace:

`https://doc.rust-lang.org/rustdoc/how-to-write-documentation.html`

CAPÍTULO 3

Elementos fundamentales del lenguaje Rust

Contenido

3.1 Tipos de datos primitivos
3.2 El tipo unidad
3.3 Tipos de datos compuestos
3.4 Tipos de datos personalizados
3.5 Cadenas de caracteres
3.6 Criterios para los nombres de los identificadores
3.7 Mutabilidad
3.8 Obtener el tipo de una variable
3.9 El operador de asignación
3.10 Otros operadores
3.11 Funciones de los tipos en coma flotante
3.12 Bifurcaciones
3.13 Bucles
3.14 Funciones
3.15 *Traits*

La sintaxis del lenguaje Rust es similar a la de C o Java, con algunas diferencias que se irán indicando a lo largo del documento.

Rust dispone de una amplia colección de tipos de datos. En este texto se hará una introducción los siguientes:

1. *Tipos primitivos: números enteros, números en coma flotante, valores booleanos, caracteres y el tipo unidad.*
2. *Tipos compuestos: arrays, vectores y tuplas.*
3. *Cadenas de caracteres.*
4. *Tipos personalizados: estructuras y enumeraciones.*

Los tipos de datos básicos son similares a los de cualquier otro lenguaje, aunque Rust ofrece tipos enteros de 16 bytes, que no están disponibles en todos los lenguajes y, para los caracteres, se utilizan valores unicode. También es característico el tipo unitario, utilizado en expresiones que no devuelven ningún valor.

Los tipos compuestos, arrays y tuplas, en principio son similares a las que se pueden encontrar en otros lenguajes, si bien su conexión con los iteradores les confieren mayores posibilidades. Existen diferentes tipos para colecciones. Se tratarán los vectores, por ser los más sencillos y más habituales.

Hay varios tipos de datos para operar con cadenas de caracteres. Se utilizan valores unicode, lo que permite trabajar en cualquier idioma, incluso a nivel de nombres de identificadores.

Las diferencias con otros lenguajes de programación son más evidentes en los tipos de datos personalizados: las estructuras y las enumeraciones. En Rust se trata de tipos de datos muy potentes que permiten incorporar funciones asociadas y, en el caso de las enumeraciones, unos patrones de coincidencia que posibilitan la utilización de técnicas de programación funcional.

También se tratarán en este capítulo conceptos como la asignación de variables, la declaración de mutabilidad, las construcciones habituales para bifurcaciones, bucles y la sintaxis básica para definir funciones.

3.1 Tipos de datos primitivos

3.1.1 Tipos para números enteros

Rust admite números enteros de 8, 16, 32, 64 y 128 bits, con signo o sin signo. Los tipos con signo se declaran poniendo una letra “i” y el número de bits. Los números sin signo se declaran poniendo la letra “u” y el número de bits. Además existen otros dos tipos denominados *isize* y *usize* que, según la plataforma para la que se esté desarrollando, tendrán un tamaño u otro. Se corresponden con el tamaño de los punteros en la plataforma. Por ejemplo, en plataformas con procesador de 32 bits, tienen 4 bytes de tamaño y, en plataformas de 64 bits, tienen un tamaño de 8 bytes.

Cuando se escribe un número entero literal, sin especificar ninguna longitud de bits, se interpreta que es un número del tipo *i32*. Los *i32* son números enteros que ocupan 4 bytes en memoria. Su valor está entre -2147483648 y 2147483647

(entre menos dos mil millones y más dos mil millones, aprox). Es el tipo de número entero usado habitualmente por lenguajes como Java, por ejemplo.

El siguiente ejemplo declara un número entero *i32* y lo escribe en pantalla. A continuación, escribe el valor del mínimo número *i32* y de máximo número *i32*:

```
let n = 127;
println!("El valor de n es {}", n);
println!("Mínimo:{} Máximo:{}", i32::MIN, i32::MAX);
```

Observe la primera línea de código dentro de la función *main()*:

```
let n = 127;
```

Esta línea de código crea una variable de nombre *n* y le asigna el valor 127. La cláusula *let* se utiliza para declarar una variable. En este caso, se crea la variable *n* y, en la misma línea de código, se le asigna el valor *127*. El signo = es el *operador de asignación*: asigna el resultado de la expresión que aparece a la derecha del operador a la variable cuyo nombre aparece a la izquierda del operador. Más adelante se hablará más sobre creación y asignación de variables. La línea de código termina con *punto y coma*.

En la línea de código que se ha analizado en el párrafo anterior, como no se ha especificado ningún tipo de datos y el valor literal es un entero, Rust interpreta que la variable *n* es del tipo *i32*, entero de 4 bytes.

Las siguientes líneas de código utilizan la macro *println!()*. El funcionamiento de esta macro es similar al de la instrucción *printf()* de los lenguajes Java o C: en la cadena entrecomillada se ponen especificaciones de formato que se sustituirán en la salida por el valor de las variables que aparecen a continuación de la cadena de caracteres. En este caso, la línea de código es:

```
println!(“El valor de n es {}”, n);
```

La cadena de caracteres tiene un texto y una pareja de llaves. La pareja de llaves es la forma de especificar en Rust el formato de salida. En esa posición se incrustará el valor de la variable que aparece a continuación de la cadena de caracteres, la variable *n*. Más adelante se verán formas de alterar el formato de salida. Si solo se ponen las dos llaves, el formato de salida será el que tiene definido Rust por defecto. Una vez más, la línea de código se termina con punto y coma.

La siguiente línea de código es:

```
println!("Mínimo:{} Máximo:{}", i32::MIN, i32::MAX);
```

En este caso, la cadena de caracteres de la macro *println!()* tiene dos especificaciones de formato, dos parejas de llaves. Ahí se incrustará el valor de las variables que aparecen a continuación, que son *i32::MIN* e *i32::MAX*. Estas variables corresponden a los valores mínimo y máximo de los números del tipo *i32*. Son constantes predefinidas en la librería de Rust.

Observe cómo se accede a ellas: se pone el nombre del tipo de datos, dos veces dos puntos y el nombre de la constante. En Rust, *dos veces dos puntos* es la forma de acceder a los valores estáticos de las librerías o de los tipos de datos definidos por el usuario. Java, por ejemplo, utiliza un solo punto para acceder a los valores. En Java, para acceder al valor mínimo del tipo *int*, se habría escrito así:

```
Integer.MIN_VALUE
```

En Java, la constante se llama *MIN_VALUE* y es un atributo estático de la clase *Integer*. En Java, para acceder a los métodos o atributos estáticos de una clase se utiliza el nombre de la clase, un punto y el nombre del atributo. En Rust, para acceder a los métodos o atributos estáticos se utilizan dos veces dos puntos. Se verá más adelante que para acceder a los métodos o atributos *de instancia* se utiliza un solo punto, como en Java.

Cuando se quiere utilizar una variable de un tipo entero que no sea el *i32*, hay que declarar explícitamente el tipo de datos. Para declarar que una variable es de un tipo de datos concreto se pueden utilizar dos procedimientos:

- Declarando el tipo de datos a continuación del nombre de la variable, separado por dos puntos. Por ejemplo, la siguiente línea de código declara y asigna valor a una variable de nombre *x* y del tipo *u8*, o sea un entero sin signo de un byte de tamaño:

```
let x:u8 = 255;
```

- También se puede especificar el tipo de datos escribiendo el nombre del tipo a continuación del valor literal, sin espacios. La siguiente línea de código sería equivalente a la anterior:

```
let x = 255u8;
```

3.1.2 Números en coma flotante

Rust ofrece dos tipos de datos primitivos para números en coma flotante: *f64* y *f32*. Los valores del tipo *f64* utilizan 8 bytes de almacenamiento en memoria; es el equivalente del tipo *double* de otros lenguajes. Los valores del tipo *f32* utilizan 4 bytes para el almacenamiento; es el tipo equivalente al *float* de otros lenguajes. Como se sabe, los números *f64* ofrecen una precisión aproximada de 15 decimales y los números *f32* ofrecen una precisión aproximada de 8 decimales.

Las siguientes serían declaraciones y asignaciones válidas de variables de los tipos *f64* y *f32*:

```
let a = 3.141592;
let b = 2.1e-3;
let c = 3.14f32;
let d: f32 = 3.1e5;
println!("{} {} {} {}", a, b, c, d);
```

El separador de decimales que se utiliza en los literales es el punto. Si se asigna a una variable un valor literal que incluya el punto decimal y no se especifica otra cosa, Rust interpreta que la variable es del tipo *f64*. Se puede utilizar la notación exponencial para escribir números literales.

Rust ofrece varias constantes predefinidas para números *f64* o *f32*. El siguiente código muestra algunos ejemplos:

```
let a = std::f64::consts::PI;
let b = std::f32::consts::E;
let c = std::f64::consts::SQRT_2;
println!("{:.12} {:.6} {:.2}", a, b, c);
```

Observe que se ha utilizado la especificación de formato “{:.n}” para fijar el número de decimales que se muestran en la salida.

Hay dos juegos independientes de constantes para valores *f64* y *f32*. Se pueden consultar en los siguientes enlaces:

`https://doc.rust-lang.org/std/f64/consts/index.html`

`https://doc.rust-lang.org/std/f32/consts/index.html`

3.1.3 Valores booleanos

El tipo de datos se denomina *bool*. Las variables del tipo bool pueden tomar el valor *true* o *false*. No se pueden utilizar el número 1 o el número 0 para sustituir a las palabras *true* o *false*.

El siguiente código podría servir para declarar una variable del tipo *bool*:

```
let si = true;
```

3.2 El tipo unidad

El *tipo unidad* o *tupla vacía* se representa por "()" y sirve para indicar que una expresión devuelve un valor vacío. En Rust, es importante la distinción entre *expresiones* y *declaraciones*. Las declaraciones son instrucciones o líneas de código que no devuelven ningún valor, mientras que las expresiones son instrucciones o líneas de código que devuelven algún valor. Como se irá comprobando a lo largo del curso, Rust utiliza preferentemente expresiones que devuelven valores, aunque puede suceder que el valor devuelto sea el valor unidad o valor vacío "()".

3.2.1 Caracteres

El tipo utilizado para almacenar caracteres individuales el el tipo *char*. Cada carácter es un valor *Unicode*, por lo que es posible guardar no solo caracteres, sino todo tipo de símbolos. Una consecuencia es que un carácter puede ocupar más de un byte. El siguiente ejemplo muestra cómo usar valores *unicode*:

```
let a = 'a';
let epsilon = '\u{0190}';
let esqui = '\u{26f7}';
let emoticono = '\u{1f601}';
println!("{} {} {} {}", a, epsilon, esqui, emoticono);
```

3.3 Tipos de datos compuestos

3.3.1 Arrays

En programación, un *array* es un tipo de datos que agrupa en una misma entidad una colección de valores del mismo tipo de datos que se guardan en memoria en posiciones consecutivas; se puede acceder a cada valor individual utilizando un índice. A cada valor de un array se le suele denominar *elemento* o *componente* del array.

En Rust, los arrays tienen que tener un tamaño y un tipo de datos conocidos en tiempo de compilación. Una vez creado un array, se puede acceder a los elementos con la notación de índices entre corchetes, siendo el primer elemento el de índice 0. El siguiente código declara algunos arrays e imprime algunos elementos individuales:

```
let a = [1, 2, 3, 4, 5];
println!("{}", a[2]); // Imprime 3
let b: [f32;3]; // Declara un array de 3 elementos del tipo f32
let c: [u8;3] = [0, 101, 255]; // Crea array de 3 u8
println!("{:?}", c); // Imprime [0, 101, 255]
```

Observe, en la última línea de código, la especificación de formato "{:?}". Algunos tipos de datos, como es el caso de los arrays, se pueden imprimir utilizando dicha especificación de formato[1].

3.3.2 Tuplas

Las tuplas son un tipo de datos que une en una sola entidad varios elementos que no tienen por qué ser del mismo tipo de datos. Se declaran con paréntesis (no corchetes como los arrays). Al igual que sucede con los arrays, hay que definir en tiempo de compilación el tamaño y el tipo de datos de cada una de las componentes de la tupla. Una vez creada una tupla, no se puede cambiar su tamaño ni el tipo de datos de sus componentes. Los elementos individuales pueden ser accedidos con la notación *variable-punto-índice*, como se hace en el siguiente código:

```
let a: (u8, u8, u8) = (255, 127, 255);
println!("{}", a.1); // Imprime 127
```

3.3.3 Vectores

Los vectores son similares a los arrays pero, a diferencia de estos, pueden cambiar de tamaño de manera dinámica (en tiempo de ejecución). Todos los elementos de un vector tienen que ser del mismo tipo de datos. Para poder modificar las componentes o para añadir o eliminar componentes de un vector, será necesario

[1] La especificación de formato "{:?} se puede utilizar con los tipos de datos que implementan el *trait Debug*. A este formato se le suele denominar *pretty-print*.

declararlo *mutable*. Más adelante se explicará cómo hacerlo. Se puede acceder a los elementos individuales con notación de índice entre corchetes.

El siguiente ejemplo crea un vector de 3 componentes del tipo *f64* utilizando la macro *vec![]*, lo imprime en pantalla y luego imprime uno de sus elementos:

```
let v = vec![3.14, 6.28, 3.14];
println!("{:?}", v); // Imprime [3.14, 6.28, 3.14]
println!("{}", v[1]);
```

El tipo de datos es *Vec<T>*. Se trata de un tipo de datos parametrizado en el que *T* representa el tipo de datos de las componentes del vector. En Rust, es habitual denominar a *T* un *tipo genérico*. El tipo *Vec<T>* ofrece numerosos métodos utilitarios que se pueden consultar en la documentación de la librería estándar, en la siguiente dirección:

`https://doc.rust-lang.org/std/vec/struct.Vec.html`

3.4 Tipos de datos personalizados

3.4.1 Estructuras

Las estructuras definen un tipo de datos que incorpora en la misma variable varias componentes de distintos tipos. Se puede acceder al valor individual de un campo de una estructura mediante la notación *variable-punto-nombre_campo*

El siguiente ejemplo crea una estructura *Point*, con dos campos *x* e *y* del tipo *i32*. En el programa principal se crea una instancia de dicha estructura y se imprime en pantalla el valor de sus campos:

```
struct Point {
   x: i32,
   y: i32
}
fn main() {
   let origen = Point{x: 0, y: 0};
   println!("{} {}", p.x, p.y); // Imprime 0 0
}
```

Observe la sintaxis para crear la instancia *p* de la estructura, poniendo entre llaves los nombre de campos y los valores que se asignan a los mismos.

Se pueden definir funciones asociadas al tipo de datos, lo que convierte a las estructuras de Rust en un tipo de datos similar a las clases de otros lenguajes. Más adelante, cuando se expliquen las funciones, se explicará cómo crear funciones asociadas a las estructuras.

Las estructuras descritas contienen campos con nombre, pero también es posible crear estructuras con campos sin nombre, que se suelen denominar *estructuras tuplas*, como se hace en el siguiente ejemplo:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
   let black = Color(0, 0, 0);
   let origin = Point(0, 0, 0);
}
```

En este caso, las instancias de *Point* y de *Color* son distintos tipos de datos, aunque los dos consten de tres números *i32*. No se puede utilizar una variable de un tipo en sustitución de una variable del otro tipo. Para acceder a los campos, se utiliza el nombre de variable, un punto y el índice que ocupa, como en las tuplas:

```
let x = origin.0;
```

Otra variante es la denominada *estructuras unidad*, que no tienen campos. Pueden ser útiles para depuración de código o en otros contextos en los que solo se necesita *marcar* la existencia de un tipo sin datos:

```
struct SinCampos;

fn main() {
   let x = SinCampos;
}
```

3.4.2 Enumeraciones

En programación, las enumeraciones son un tipo de datos que agrupa en una misma entidad un conjunto de valores con nombre. A cada uno de los valores que componen la enumeración se le denomina *variante* o *miembro* de la enume-

ración. Las variables de dichos tipos pueden tomar solo uno entre los valores de las variantes definidas. En un programa, se asocian variables a dichas variantes, permitiendo la comparación entre ellas.

En algunos lenguajes, las enumeraciones son simplemente una forma de dar nombre a algunas constantes, asociando cada variante con un número entero, por ejemplo. Esto permite hacer comparaciones entre distintas variables. El siguiente código muestra un ejemplo de utilización en C:

```
enum Level {
   LOW,
   MEDIUM,
   HIGH
};
int main() {
   enum Level nivel = MEDIUM;
   if (nivel == LOW) {
      // haz_una_cosa()
   } else {
      // haz_otra_cosa()
   }
}
```

Combinando este tipo de enumeraciones con las estructuras, sería posible distinguir unas instancias de otras. Otros lenguajes, como Java, añaden algunas capacidades extra a las enumeraciones, convirtiéndolas en un tipo especial de clase de objetos con propiedades y métodos asociados.

En Rust, las enumeraciones son un tipo de datos muy potente que permite asociar un valor de cualquier tipo de datos a cada una de las variantes de la enumeración o asociar funciones al tipo de datos, de manera similar a lo que se hace con las estructuras.

Rust utiliza dos enumeraciones especiales, *Result* y *Option*, para gestionar los errores de ejecución o los valores devueltos por las funciones. Estas dos características, en combinación con la instrucción *match*, proporcionan a Rust algunos patrones de diseño característicos de la programación funcional, como se irá viendo.

La sintaxis de la declaración de una enumeración en Rust es similar a la de C o Java. El siguiente ejemplo declara un tipo de enumeración y crea un valor de

dicho tipo.

```
enum Color {
   Rojo,
   Verde,
   Azul
}
fn main() {
   let color = Color::Verde;
}
```

Al tipo, en este caso *Color*, se le denomina *identificador* o simplemente *tipo*. A cada uno de los posibles valores que pueden tomar las variables, *Rojo*, *Verde* o *Azul*, se les denomina *variantes* o *miembros* del tipo *Color*, como ya se ha indicado.

Para poder realizar comparaciones entre variables del tipo creado, se debe indicar al compilador que implemente para dicho tipo el *trait PartialEq*, lo que equivale a imponer que se pueden utilizar los operadores *==* y *!=*. Esto se consigue escribiendo una anotación `#derive[PartialEq]` en el tipo de datos creado. Con esto, sería posible hacer una comparación entre dos variables del tipo *Color*, como se hace en el siguiente código:

```
#[derive(PartialEq)]
enum Color {
   Rojo,
   Verde,
   Azul
}

fn main() {
   let color_1 = Color::Rojo;
   let color_2 = Color::Azul;
   if color_1 == color_2 {
      // haz_algo()
   }
}
```

Al compilar el código anterior, se muestra un mensaje de advertencia indicando que la variante *Color::Verde* no se utiliza nunca en el código.

Cada variante de una enumeración puede llevar asociado un valor de cualquier tipo de datos, que además puede ser diferente para cada variante.

El siguiente ejemplo utiliza una enumeración para definir un color. La variante *RGB* define un color mediante sus proporciones de *Red*, *Green* y *Blue*; la variante *RGBA* añade una componente con el nivel de transparencia *Alpha* y la variante *CMYK* combina *Cyan*, *Magenta*, *Yellow* y *Key* (*Negro*).

```
#[derive(PartialEq)]
enum Color {
   RGB(u8, u8, u8),
   RGBA(u8, u8, u8, u8),
   CMYK { cyan: u8, magenta: u8, yellow: u8, key: u8 },
}
```

En el código anterior, se utilizan dos maneras diferentes de asociar valores a las variantes del tipo *Color*. En las variantes *RGB* y *RGBA*, se ha asociado una tupla de enteros tipo *u8*, en un caso con tres valores y en el otro con cuatro. En cambio, la variante *CMYK* utiliza una estructura de campos con nombre. En un caso real, lo lógico sería utilizar un mismo criterio con todas las variantes del tipo.

Definido el tipo *Color*, se podrían crear variables y hacer comparaciones:

```
#[derive(PartialEq)]
enum Color {
   RGB(u8, u8, u8),
   RGBA(u8, u8, u8, u8),
   CMYK { cyan: u8, magenta: u8, yellow: u8, key: u8 },
}

fn main() {
   let alice_blue = Color::RGB(240, 248, 255);
   let naranja = Color::RGBA(243, 156, 18, 255);
   println!("{}", alice_blue==naranja); // false
}
```

También es posible utilizar los valores asociados a cada variante, utilizando instrucciones *match*[2]:

```
match alice_blue {
   Color::RGB(red, green, blue) => {
      println!("Red:{red} Grren:{green} Blue:{blue}");
   },
   _ => println!("No es RGB")
}
```

Se pueden definir funciones asociadas al tipo de datos. El siguiente ejemplo implementa una función que imprime los valores asociados a cada color:

```
impl Color {
   pub fn to_hex(&self) -> String {
      match self {
         Color::RGB(red, green, blue) => {
            format!("#{:X}{:X}{:X}", red, green, blue)
         },
         _ => format!("No implementado"),
      }
   }
}
```

3.5 Cadenas de caracteres

Las cadenas de caracteres tienen un tratamiento especial en todos los lenguajes de programación y, en general, un tratamiento diferente en cada lenguaje.

Como se ha indicado, en Rust los caracteres individuales son valores Unicode y las cadenas de caracteres también están formadas por valores Unicode.

Se usan dos tipos de datos para cadenas de caracteres:

- **&str:** se usa para los *literales entrecomillados*. Cuando en el código de un programa se asigna a una variable un literal entrecomillado, el tipo asignado es *&str*, que no está afectado por las reglas de propiedad que se verán

[2]La instrucción *match* se explicará en el Apartado 3.12. El lector necesitará comprender dicho apartado para comprender el ejemplo que se muestra.

más adelante, aunque sí que está afectado por las reglas de *vida útil* que afectan a las referencias.
- **String:** es el tipo de datos que se utiliza para las cadenas de caracteres propiamente dichas. Es un tipo afectado por las reglas de propiedad.

Las variables del tipo *&str* se crean asignando un valor entre comillas dobles:

```
let nombre = “Ramón Requena”;
```

Las variables del tipo *String* disponen de método para su creación a partir de un literal entrecomillado:

```
let nombre = String::from(“Ramón Requena”);
```

En Rust, el tipo *String* es muy potente y con una gran cantidad de métodos utilitarios. Se puede consultar su documentación en el siguiente enlace:

```
https://doc.rust-lang.org/std/string/struct.String.html
```

3.6 Criterios para los nombres de los identificadores

Los nombres válidos de los identificadores deben seguir la norma *Unicode Standard Annex #31* [9]. Se admiten caracteres Unicode, por lo que es posible utilizar la letra *ñ* u otros caracteres de otros idiomas. También está permitido usar el guion bajo, el símbolo de dólar u otros. No obstante, para facilitar la internacionalización del código, se recomienda utilizar solo los caracteres del alfabeto inglés, en mayúsculas o minúsculas, números y el guion bajo.

Además, se siguen las siguientes reglas:

- El lenguaje distingue entre mayúsculas y minúsculas, por lo que la variable *X* mayúscula es diferente de la variable *x* minúscula.
- Los números pueden formar parte del nombre de un identificador, pero no pueden ser el primer carácter del mismo.
- El guion bajo sí que puede ser el primer carácter del identificador, pero es costumbre que los nombre que comienzan con guion bajo se utilicen para identificadores que no se van a utilizar posteriormente.

En Rust, es costumbre utilizar para los identificadores de variables, nombres de funciones y otros el criterio llamado *snake case*, que consiste en utilizar palabras en minúsculas separadas por guiones bajos. Ejemplos de la utilización de este criterio podrían ser los siguientes:

x min_value get_sum()

Cuando se trata de constantes, es habitual utilizar el llamado *screaming snake case*, en el que las palabras se escriben con todas las letras en mayúsculas, por ejemplo:

VALOR_RESIDUAL SCREAMING_SNAKE_CASE

Los nombres de los tipos de datos, como los nombres de estructuras o enumeraciones, en cambio, utilizan el criterio *camel case*, con las palabras en mayúsculas y sin espacios ni guiones bajos entre ellas, por ejemplo:

Color LineaDePuntos

Aunque se podría utilizar cualquier otro criterio en los nombres de los identificadores, la utilización de este código de conducta facilita la lectura del código por distintos programadores.

Si necesita más información al respecto, puede consultar la referencia del lenguaje en la siguiente dirección:

`https://doc.rust-lang.org/reference/identifiers.html`

3.7 Mutabilidad

En Rust, las variables son *inmutables* por defecto. Una vez que se ha asignado un valor, no es posible modificarlo. El código siguiente no compila, la variable *x* es inmutable y no es posible asignarle un nuevo valor:

```
fn main() {
   let x = 8.5;
   x = 3.4; // ¡ERROR, no compila, x es inmutable!
}
```

Para que una variable sea *mutable*, hay que indicarlo explícitamente en la declaración de la variable utilizando la cláusula *mut*, como se hace en el siguiente código:

```
fn main() {
   let mut v = vec![1, 2, 3];
   v[1] = 100;
   v.push(200);
}
```

Una cosa diferente es redefinir una variable ya existente utilizando *let*, lo que se denomina *shadowing*. En realidad, lo que se hace es crear una nueva variable con el mismo nombre. La variable anterior queda inaccesible. La nueva variable no tiene por qué ser del mismo tipo que la original:

```
fn main() {
   let x: f64 = 3.14;
   println!("{}", x); // Imprime 3.14
   let x: u8 = 255;
   println!("{}", x); // Imprime 255
}
```

3.8 Obtener el tipo de una variable

La librería estándar de Rust ofrece una función que permite obtener el nombre del tipo de datos de una variable. Se muestra su uso con un ejemplo:

```
let v = vec![1, 2, 3, 4, 5];
println!("{}", std::any::type_name_of_val(&v));
// Imprime alloc::vec::Vec<i32>
```

3.9 El operador de asignación

Hay un operador fundamental en cualquier lenguaje que ya se ha utilizado anteriormente, dando por hecho que el lector tiene una comprensión, al menos intuitiva, de su funcionamiento: el *operador de asignación*.

En Rust, el operador de asignación utiliza el símbolo = (igual). A la derecha del símbolo igual se escribe una expresión, esto es, código que devuelve un resultado; a la izquierda del operador se pone el nombre de una variable. Lo que hace el operador es asignar a la variable cuyo nombre aparece a la izquierda del signo igual el resultado de la expresión que aparece a la derecha del mismo.

La línea de código de una asignación tiene que terminar en punto y coma.

El ejemplo más sencillo de expresión sería un literal, por ejemplo un número. De esta forma, el operador asignaría el valor de dicho número a la variable cuyo nombre figura a la izquierda del operador. Observe el siguiente ejemplo:

```
v = 5;
```

A la izquierda figura el nombre de una variable llamada *v* y, tras el signo igual, el número 5. Como resultado de la aplicación del operador, el valor 5 se asignará a la variable *v*.

Para poder utilizar una línea de código como la anterior, es necesario que la variable *v* haya sido declarada con anterioridad, con una instrucción *let* o con una instrucción *let mut*. En el primer caso, la asignación solo puede hacerse una vez; en el segundo caso, la asignación se puede hacer más de una vez. Es habitual hacer la declaración de la variable y la asignación de valor en la misma línea de código:

```
let v = 5;
```

Un detalle imprescindible en toda asignación es que el resultado de la expresión que aparece a la derecha del operador tiene que ser del mismo tipo de datos que la variable a la que se le asigna. En este caso, el tipo de datos es *i32* y queda establecido en el valor del literal; en otros casos, será necesario establecer el tipo de datos, bien en la declaración de la variable o bien en el propio literal, como se explicó en el Apartado 3.1.1.

Las expresiones que aparecen a la derecha del operador de asignación son cualquier código que devuelva un resultado. En la mayoría de los lenguajes de programación, las expresiones están formadas por una combinación de literales, operadores e identificadores de otras variables o funciones. En Rust, hay construcciones del lenguaje que también son expresiones y devuelven un resultado, como los bloques de instrucciones del tipo *if* o los bloques correspondientes a los bucles (*loop*, *for* y *while*). Más adelante se verán ejemplos de estos usos del operador de asignación.

Otro caso de asignación que hay que tener en cuenta es cuando la expresión que aparece a la derecha del operador de asignación es el nombre de otra variable. En ese caso, según el tipo de datos de las variables, hay dos comportamientos posibles. Como se verá en el Apartado 4.1, hay tipos de datos que funcionan con semántica *Copy* y otros tipos de datos que funcionan con semántica *Move*.

En las asignaciones (y también en el paso de parámetros a funciones), los tipos que funcionan con *Copy* realizan una copia del valor de una variable en la otra, quedando dos variables útiles, con valores iguales. Esto sucede con los tipos primitivos, los arrays y las tuplas cuyos elementos son tipos primitivos y cualquier otro tipo de datos que implemente el *trait Copy*. El siguiente sería un ejemplo utilizando números *f64*:

```
let x: f64 = 0.75;
let y = x;
assert_eq!(x, 0.75);
assert_eq!(y, 0.75);
```

Tras la asignación, las dos variables *x* e *y* siguen existiendo.

En cambio, si las variables son de un tipo que funciona con *Move*, tras la asignación la variable original deja de ser accesible y la propiedad del valor pasa a la nueva variable:

```
let v = vec![1, 2, 3];
let w = v;
assert_eq!(w, vec![1, 2, 3]);
assert_eq!(v, vec![1, 2, 3]); // ERROR, v ya no es accesible
```

3.10 Otros operadores

Además del operador de asignación, hay otros operadores que también son habituales en cualquier lenguaje de programación:

- Operadores aritméticos.
- Operadores relacionales o de comparación.
- Operadores lógicos.
- Operadores de asignación compuestos.

En todos los casos, estos operadores representan una forma abreviada de utilizar ciertas funciones que reciben uno o dos argumentos y devuelven un resultado calculado en base al valor de dichos argumentos. Los argumentos se suelen llamar también *operandos*.

En Rust, el tipo de datos de los operandos tiene que ser el mismo. No se pueden operar valores de diferentes tipos de datos. El resultado puede ser del mismo tipo que los operandos o no, según el operador que se esté utilizando.

3.10.1 Operadores aritméticos

Los operadores aritméticos utilizan dos argumentos de tipo numérico y devuelven como resultado un valor numérico del mismo tipo de datos. Se resumen en la Tabla 3.1.

Tabla 3.1: Operadores aritméticos

Operador	Significado
$+$	Suma
$-$	Resta
$*$	Producto
$/$	División
$\%$	Resto de la división

Como se ha dicho, los dos operandos tienen que ser del mismo tipo de datos y el resultado también lo será. No es posible, or ejemplo, sumar un entero *i32* con un entero *u8*:

```
let x : i32 = 12;
let y : u8 = 5;
println!("{}", x+y); // ERROR, no son del mismo tipo
```

Rust ofrece la cláusula *as* para hacer *casting* de unos tipos de datos a otros:

```
let x : i32 = 12;
let y : u8 = 5;
println!("{}", x as u8 + y); // Imprime 17
println!("{}", x + y as i32); // Imprime 17
```

En una expresión en la que se combinen varios operadores, el interprete del lenguaje opera de izquierda a derecha, pero hay elementos que alteran el orden de las operaciones. Por ejemplo, las expresiones que estén encerradas entre paréntesis, las llamadas a funciones o los nombre de otras variables, se ejecutan antes y su resultado se sustituye en la expresión. También se ejecutan antes los productos, las divisiones y el resto que las sumas y restas.

Se muestran a continuación algunos ejemplos:

$$
\begin{aligned}
&5+3+4 \quad \rightarrow \quad 8+4 \quad \rightarrow \quad 12 \\
&5+3*4-1 \quad \rightarrow \quad 5+12-1 \quad \rightarrow \quad 17-1 \quad \rightarrow \quad 16 \\
&(5+3)*4-1 \quad \rightarrow \quad 8*4-1 \quad \rightarrow \quad 32-1 \quad \rightarrow \quad 31 \\
&3+5\ \%\ 2 \quad \rightarrow \quad 3+1 \quad \rightarrow \quad 4
\end{aligned}
$$

3.10.2 Operadores relacionales o de comparación

Los operadores *relacionales*, también llamados *de comparación*, reciben dos operandos de tipo numérico y devuelven un valor *bool: true* o *false*. Se resumen en la Tabla 3.2.

Tabla 3.2: Operadores relacionales o de comparación

Operador	Significado
$<$	Menor que
$<=$	Menor o igual que
$>$	Mayor que
$>=$	Mayor o igual que
$==$	Igual (*Igual-Igual*)
$!=$	No igual

Como se puede ver en la Tabla 3.2, algunos operadores constan de dos caracteres. Merece mención especial el operador *==* (*igual-igual)*), que no hay que confundir con el operador de asignación *=* (*igual*). Esta confusión, frecuente entre estudiantes que están iniciándose en programación, da lugar a algunos *bugs*[3] difíciles de depurar.

En cuanto al operador *!=* (*no igual*), hay otros lenguajes que utilizan otra sintaxis, por ejemplo, en MATLAB se utiliza $\sim=$.

Cuando en una expresión se combinan varios operadores, se operan de izquierda a derecha. Si aparecen operadores aritméticos y operadores de comparación, los operadores aritméticos se ejecutan antes que los de comparación. A continuación se muestran algunos ejemplos:

$$5 > 2 + 3 \quad \rightarrow \quad 5 > 5 \quad \rightarrow \quad false$$
$$5 + 1 > 2 + 3 \quad \rightarrow \quad 6 > 2 + 3 \quad \rightarrow \quad 6 > 5 \quad \rightarrow \quad true$$

[3] En programación, hay distintos tipos de errores. Por una parte, están los errores de sintaxis o de codificación que hacen que el programa no compile. Suelen ser los errores más fáciles de depurar. También hay errores de ejecución que, según el tipo concreto, pueden ser más o menos difíciles de depurar. Por último están los *bugs*, errores en la lógica del programa; en los *bugs*, el programa compila y se ejecuta sin interrupciones, pero los resultados no coinciden con lo que cabría esperar. Suelen ser los más difíciles de depurar.

3.10.3 Operadores lógicos

Los operadores *lógicos* reciben uno o dos operandos del tipo *bool* y devuelven un resultado del tipo *bool*. Se resumen en la Tabla 3.3.

Tabla 3.3: Operadores lógicos

Operador	Significado
&&	AND : *true*, si los dos operandos son *true*, *false* en caso contrario
||	OR : *true*, si al menos uno de los operadores es *true*, *false* en caso contrario
!	NOT : *true*, si el operando es *false*, *false* en caso contrario

En Rust, los operadores lógicos aplican el llamado *short-cicuiting*, esto es, evalúan el primer operando y si queda resuelto el resultado, no evalúan el segundo. Por ejemplo, en un *AND*, si el primer operando es *false*, el resultado es *false* cualquiera que sea el valor del segundo operando. En el operador *OR*, si el primer operando es *true*, el resultado es *true* con seguridad.

En cierto sentido, el operador lógico *AND* se podría equiparar al operador aritmético producto y el operador lógico *OR* se podría equiparar a la suma. De hecho, si en una expresión aparecen operadores *AND* y operadores *OR*, se ejecutan primero los *AND*. El operador *NOT* tiene precedencia sobre los otros operadores lógicos: primero se ejecuta los operadores *NOT*, luego los operadores *AND* y, por último, los operadores *OR*.

Cuando en una expresión aparecen operadores lógicos y operadores de comparación, los operadores de comparación se operan antes que los lógicos (recuerde que los operadores aritméticos se ejecutan antes que los de comparación).

Un caso paradigmático de esta regla de precedencia de los operadores de comparación sobre los lógicos es la condición para indicar que un número está dentro de determinado intervalo. En notación algebraica, un ejemplo podría ser:

$$x \in (a, b) \Rightarrow a < x < b$$

La expresión algebraica $a < x < b$ encierra dos comparaciones y un operador *AND*: *a es menor que x Y x es menor que b*. En Rust, se podría poner:

```
a<x && x<b
```

El orden de precedencia de los operadores hace innecesarios los paréntesis, aunque también se podrían utilizar para facilitar la lectura.

3.10.4 Operadores compuestos de asignación

Son operadores que realizan una operación aritmética y una asignación. Se resumen en la Tabla 3.4.

Tabla 3.4: Operadores compuestos de asignación

Operador	Significado
$+=$	Suma y asignación
$-=$	Resta y asignación
$*=$	Producto y asignación
$/=$	División y asignación
$\%=$	Resto y asignación

3.11 Funciones de los tipos en coma flotante

Todos los tipos de datos que se han visto en los apartados anteriores disponen de funciones utilitarias que se pueden consultar en la documentación de la librería estándar:

```
https://doc.rust-lang.org/std/index.html
```

En este apartado se van a comentar las funciones asociadas a los tipos *f64* y *f32*, por ser de uso frecuente en cálculos matemáticos. La totalidad de los métodos se pueden consultar en los siguientes enlaces:

```
https://doc.rust-lang.org/std/primitive.f64.html

https://doc.rust-lang.org/std/primitive.f32.html
```

Lo primero que hay que indicar es que hay un juego de funciones para operar con valores *f32* y otro juego de funciones para operar con valores *f64*. Como se comentó en el caso de los operadores, no se pueden mezclar tipos, cuando se opera con valores *f64* hay que utilizar funciones del tipo *f64* y análogamente cuando se opera con valores *f32*.

Lo siguiente que suele sorprender al trabajar con funciones matemáticas en Rust es la forma de invocarlas. No se llama a una función y se le pasa el argumento entre paréntesis, se invoca con la notación *valor-punto-función*, de la misma forma que se llama a las funciones asociadas a otros tipos de datos. Por ejemplo, para calcular la raíz cuadrada de *4.0* se haría de la siguiente forma:

```
4.0f64.sqrt()
```

Observe que ha sido necesario especificar el tipo de datos concreto, *f64*, en el literal del valor *4.0*. Si no se hace, el compilador indica que hay ambigüedad entre la función *sqrt()* para *f64* y la función *sqrt()* para *f32*. Si el valor ya tuviera el tipo de datos asignado, no sería necesario hacerlo así:

```
let x: f64 = 4.0;
let y = x.sqrt();
```

Normalmente, uno está acostumbrado a trabajar con lenguajes en los que las funciones reciben los parámetros entre los paréntesis. Por ejemplo, en Java se habría escrito algo así:

```
double x = 4.0;
double y = Math.sqrt(x);
```

En un primer momento, la forma de escribir las funciones en Rust se hace un poco rara. Pero es una cuestión de cambiar la forma en la que se interpretan las expresiones anteriores. En el caso de la función *sqrt(x)*, la leeríamos como *"raíz cuadrada de x"*. En el caso de la sintaxis de Rust, *x.sqrt()*, es mejor leerlo como "al valor x se le aplica la función raíz cuadrada".

Puede parecer artificioso, pero piense en el caso de varias funciones compuestas, por ejemplo una expresión del tipo:

$$\sqrt{\sin\left(x^2\right)}$$

En realidad, la forma de operar una expresión de ese tipo es de dentro hacia afuera: se calcula el cuadrado de *x*, luego el seno de ese resultado y, finalmente, la raíz cuadrada del resultado. Se leería: "*raíz cuadrada del seno de x al cuadrado*". En Java se escribiría:

```
Math.sqrt(Math.sin(x*x))
```

En el caso de Rust, esa expresión se escribiría así:

```
(x*x).sin().sqrt()
```

Se podría leer como: "*Se calcula x al cuadrado, se le aplica la función seno y, al resultado, se le aplica la función raíz cuadrada*". En la sintaxis se recoge el orden en el que se realizan las operaciones, la forma en la que se componen las funciones que se van aplicando. Es una forma más *declarativa* de escribir las operaciones, aunque se aleje de la forma en la que estamos acostumbrados a escribir esas expresiones.

De hecho, una vez que uno se acostumbra a escribir las operaciones de esta

manera, se convierte en natural y lógica y, lo que se hace extraño y artificioso, es la forma clásica de escritura de esas mismas expresiones. En realidad, las dos formas de escribir las expresiones se corresponden con las dos formas de expresar la composición de funciones en matemáticas:

$$f(g(h(x))) \quad \rightarrow \quad (h \cdot g \cdot f)(x)$$

Por último, hacer notar cómo se encadenan varias operaciones en Rust con la notación punto. A lo largo de este escrito se verán otras ocasiones en las que se utiliza esta composición de operaciones, una tras otra, simplemente poniendo el punto.

No obstante, si el lector no consigue acostumbrarse a esta forma de escribir las funciones y prefiere a toda costa utilizar la forma tradicional de invocar funciones, siempre podría codificar las que necesite para poder invocarlas de la manera tradicional, como se hace en el siguiente programa:

```
fn main() {
   let x: f64 = 0.75;
   let y = sqrt(sin(x));
   println!("{y:.4}"); // Imprime 0.8256
}
fn sin(x: f64) -> f64 {
   x.sin()
}
fn sqrt(x: f64) -> f64 {
   x.sqrt()
}
```

3.12 Bifurcaciones

3.12.1 Bifurcaciones *if...else*

La sintaxis es similar a la de otros lenguajes. La condición tiene que ser una expresión que devuelva un valor *bool*, no se admiten números ni otros tipos de datos. Como detalle, no es necesario poner la condición entre paréntesis. El siguiente código muestra un ejemplo de *if* con cláusula *else*:

```
fn main() {
   let n = 3;
   if n < 5 {
      println!("Condición verdadera");
   } else {
      println!("Condición falsa");
   }
}
```

Es posible utilizar bifurcaciones *if* sin rama *else* y también con ramas múltiples del tipo *else if*, como se hace en el siguiente ejemplo:

```
fn main() {
   let n = 3;
   if n==0 {
      println!("Cero");
   } else if n%2==0 {
      println!("Par");
   } else {
      println!("Impar");
   }
}
```

Si bien la sintaxis es similar a la de otros lenguajes, hay una diferencia fundamental en cuanto al comportamiento de la instrucción. En Rust, la instrucción *if* es una expresión, no una declaración. Esto quiere decir que cada rama de un *if*, devuelve un valor. El valor que devuelven las dos ramas del ejemplo anterior es el valor unidad, ().

Al tratarse de una expresión, es posible usar un *if* en la parte derecha de una asignación, como se hace en el siguiente ejemplo:

```
let n = 3;
let is_par = if n%2==0 {true} else {false};
```

Las dos ramas del *if* tienen que devolver el mismo tipo de datos, si no, se produce un error de compilación. El siguiente código daría error:

```
let n = 3;
let is_par = if n%2==0 {n} else {"impar"};
```

3.12.2 Bifurcaciones *match*

La otra construcción que permite gestionar bifurcaciones es el bloque `match`, similar al `case` o al `switch` de otros lenguajes. En un bloque `match`, se evalúa un valor entre varias opciones. Se denominan *ramas*, a cada una de las líneas de bifurcación en función de los posibles valores. Los bloques `match` son bifurcaciones *en paralelo*, en las que el código sale por una rama de entre varias.

La instrucción `match` fuerza a comprobar todos los posibles valores de la variable que se está evaluando. En el siguiente ejemplo se muestra la sintaxis de la instrucción:

```
fn main() {
   let x = 10;
   let resto = x%2;
   match resto {
      0 => println!("Par"),
      _ => println!("Impar"),
   }
}
```

El símbolo del guion bajo _ que se utiliza en las ramas de `match` se interpreta como «*cualquier valor*». Es habitual escribir el símbolo _, pero se podría utilizar cualquier otro nombre de variable, como `other` o cualquier otro. El motivo de utilizar el guion bajo es que el compilador de *Rust* genera un aviso cuando se declara una variable y no se utiliza, pero este aviso no se genera si el nombre de la variable es el guion bajo _ o comienza por un guion bajo _.

La construcción `match` no solo funciona con enteros, se pueden probar valores de cualquier tipo. En el caso de los números en coma flotante, la necesidad de comprobar todos los valores posibles puede hacer necesario establecer condiciones lógicas del tipo «*menor que*» u otras similares. Para ello, se pone la condición con un `if` después de la declaración de la variable en la rama correspondiente, como se hace en el siguiente ejemplo:

```
fn main() {
   let x = 2.5;
   match x {
      valor if valor<10.0 => println!("x < 10.0"),
      valor if valor==10.0 => println!("x == 10.0"),
      _ => println!("x > 10.0"),
   };
}
```

También se puede utilizar la coincidencia de patrones con tipos personalizados, por ejemplo con una tupla:

```
match (1, "Hello") {
   (i, _) if i < 0 => println!("Negative integer: {}", i),
   (_, s) => println!("{} World", s),
}
```

Un uso muy habitual de los bloques `match` es para comprobar las opciones posibles cuando el resultado es una enumeración del tipo `enum`, como se verá en el Capítulo **??**.

3.13 Bucles

Rust tiene tres tipos de bucles: *loop*, *while* y *for*.

El bucle *loop* se ejecuta de manera indefinida, salvo que se le indique explícitamente que termine:

```
fn main() {
   loop {
      println!("Bucle infinito!");
   }
}
```

Si ejecuta el código anterior, tendrá que pulsar *CTRL + C* para terminar el programa.

Las cláusulas *break* y *continue* se pueden utilizar dentro de cualquier bucle para alterar el flujo normal del mismo. La cláusula *continue* ignora las restantes

instrucciones de la iteración e inicia una nueva iteración. La cláusula *break* ignora el resto del código en el bucle y el resto de las iteraciones que pudiera haber pendientes y salta a la siguiente instrucción de código tras el bucle.

Los bucles *while* también tienen una sintaxis similar a la de otros lenguajes, con la salvedad, si acaso, de que no es necesario encerrar la condición entre paréntesis.

```
fn main() {
   let mut contador = 0;
   while contador < 5 {
      println!("{contador}");
      contador += 1;
   }
}
```

Los bucles *for* se utilizan para recorrer colecciones. Su sintaxis es la del siguiente ejemplo:

```
fn main() {
   let a = [10, 20, 30, 40, 50];

   for elemento in a {
      println!("El valor es: {elemento}");
   }
}
```

Combinando los bucles *for* con los *Range* (*rango*), es posible generar muchos tipos de bucles con una sintaxis más declarativa que la de sus equivalentes bucles *while*. Por ejemplo, el bucle contador se podría codificar de la siguiente forma:

```
fn main() {
   for contador in 0..4 {
      println!("{contador}");
   }
}
```

El código resultante, no solo es más declarativo que su equivalente *while*, además no es necesario hacer mutable la variable contador.

Los rangos implementan el *trait Iterator* y otros, lo que les proporciona numerosos métodos de utilidad que pueden servir para definir distintos tipos de bucles. Por ejemplo, el siguiente ejemplo usa los números del rango en orden inverso y de dos en dos:

```
fn main() {
   for contador in (0..8).rev().step_by(2) {
      println!("{contador}");
   }
}
```

En Rust, los bucles también son expresiones. Para que un bucle devuelva un valor, se utiliza la cláusula *break*. El valor devuelto será el resultado de la expresión que aparezca a continuación del *break*.

```
   fn main() {
      let mut contador = 0;
      let result = loop {
         contador += 1;
         if contador == 10 {
            break contador * 2;
         }
      };
      println!("{result}"); // Imprime 20
   }
```

La línea de código donde aparece *break* se puede acabar con punto y coma

o sin él, pero observe el punto y coma que cierra la asignación a *result*. Ahí sí es necesario el punto y coma, pues se trata de una asignación.

3.14 Funciones

De manera similar a la de otros lenguajes, las funciones se definen mediante una línea de signatura seguida del cuerpo de la función entre llaves. La signatura utiliza la cláusula *fn* seguida de un nombre válido de función, la lista de parámetros entre paréntesis y, si devuelven algún valor, se pone una *flecha* "->" seguida del tipo de datos del valor devuelto. A continuación, entre llaves, irá el cuerpo de la función.

```
fn nombre(param_1: tipo_1, ...) -> tipo_devuelto {
   // Instrucciones del cuerpo de la función
}
```

Para devolver un valor dentro del cuerpo de la función, no es necesario utilizar la cláusula *return*, aunque puede usarse; es suficiente dejar una expresión sin terminar en punto y coma y el resultado de dicha expresión se utilizará como valor devuelto. Solo se puede devolver un valor, aunque puede ser de cualquier tipo, incluyendo arrays, estructuras u otros.

Un ejemplo podría ser el siguiente:

```
fn por_dos(x: i32) -> i32 {
   2*x
}
```

En el ejemplo anterior, la función se llama *por_dos*, recibe un parámetro de nombre *x* del tipo *i32* y devuelve un valor del tipo *i32*. El cuerpo de la función es una sola línea de código que multiplica por 2 el valor de *x* que recibe. Esa línea de código se finaliza sin punto y coma, por lo que el resultado de la expresión *2*x* será el valor devuelto por la función.

3.14.1 Funciones asociadas a las estructuras y enumeraciones

Cuando se implementan funciones asociadas a una estructura o una enumeración, se hace dentro de un bloque *impl*. Hay dos tipos de funciones asociadas a las estructuras y enumeraciones:

- *Estáticas:* asociadas al tipo de datos, no a las instancias de dicho tipo. Se invocan usando el nombre del tipo, dos veces dos puntos y el nombre de la función.
- *De instancia:* el primer parámetro debe ser la palabra clave *self*, que identifica a la instancia concreta que invoca al método. El parámetros *self* se puede especificar de tres formas: *self*, *&self* y *&mut self*, según se pase la propiedad, una referencia al valor o una referencia mutable al valor. Se ira explicando más adelante.

El siguiente ejemplo, crea la estructura *Point* e implementa dos métodos: uno estático llamado *new()*, que hace las veces de constructor y otro de instancia, llamado *sum()*, que devuelve la suma de las componentes del *Point*. En el programa principal se crea una instancia de *Point* y se invocan ambos métodos:

```
struct Point {
   x: i32,
   y: i32
}
```

```
impl Point {
   fn new(x: i32, y: i32) -> Point {
      Point{x: x, y: y}
   }
   fn sum(&self) -> i32 {
      self.x + self.y
   }
}
fn main() {
   let p = Point::new(3, 2); // Crea una instancia de Point
                             // y la asigna a la variable p
   let suma = p.sum();       // Suma las componentes de p
   println!("{}", suma);     // Imprime 5
}
```

Observe que, en el método *sum()*, se utiliza para el primer parámetro la forma *&self*, que es una referencia a la instancia que invoca el método, para no consumir la propiedad del mismo. Es la forma habitual de hacerlo y se comprenderá mejor cuando se explique el concepto de propiedad.

Observe también que, en el método *new()*, el nombre de los parámetros coincide con el nombre de los campos de *Point*. En estos casos se podría omitir el nombre del campo en la asignación de valores, haciendo:

```
fn new(x: i32, y: i32) -> Point {
   Point{x, y}
}
```

Observe, por último, la forma de invocar los dos métodos: el método estático con el nombre del tipo y dos veces dos puntos y el método de instancia con el nombre de variable y un solo punto.

En el caso de las enumeraciones, la forma de implementar métodos es igual que en el caso de las estructuras.

3.14.2 Closures

Son funciones, frecuentemente anónimas, declaradas *inline* dentro del contexto de otra función y que pueden acceder a las variables de dicho contexto.

Las closures utilizan dos barras verticales, de la misma forma que las funciones ordinarias utilizan los paréntesis. Entre las barras verticales se ponen los nombres de los parámetros. En la mayoría de los casos, el compilador es capaz de inferir el tipo de los parámetros y no es necesario especificarlo. A continuación, se pone el código de la closure que, si solo consta de una línea, no es necesario encerrarlo entre llaves.

El siguiente ejemplo muestra una closure:

```
let suma_uno = |x| x+1;
println!("{}", suma_uno(5)); // Imprime 6
```

En el ejemplo, se declara una closure y se asigna a la variable *suma_uno*; se trata de una closure que recibe un argumento *x* y devuelve *x+1*.

Esta inferencia del tipo de datos de los parámetros en las closures puede permitir desarrollar closures para tipos genéricos. El siguiente código define una closure que recibe un parámetro *x*. Al no especificar el tipo de datos de *x*, la closure se puede utilizar con un valor entero o con una cadena de caracteres. Eso sí, el primer uso que se haga de la closure definirá el tipo de datos de *x* y ya

solo se podrá usar la closure con ese tipo de datos.

```
fn main() {
   let f = |x| x;
   let s = f(String::from("hello"));
   let n = f(5); // ERROR, x ha quedado fijado a  tipo String
}
```

En el código anterior, el primer uso que se haga de la closure fijará el tipo de datos del parámetro para todos los usos subsiguientes. Si se hubiera usado primero con un valor entero, el uso posterior con una cadena de caracteres habría resultado en un error de compilación.

Una característica de las closures es que pueden acceder a las variables del contexto en el que ha sido definida la closure. Según la forma en la que se realice dicho acceso, Rust establece tres tipos de closures:

- Acceso por referencia: *&T*. Da lugar a closures del tipo *Fn*.
- Acceso por referencia mutable: *&mut T*. Son closures del tipo *FnMut*.
- Acceso por valor: *T*. Según el tipo de datos de *T*, la closure puede tomar la propiedad del mismo. En este caso la closure es del tipo *FnOnce*.

En cada caso, Rust trata de asignar a la closure el tipo menos restrictivo que sea posible. Toda función es al menos *FnOnce*.

Hay que tener en cuenta que, para acceder a una variable del contexto, no es necesario especificar su nombre como parámetro de la closure. Los parámetros formales que se indican entre las barras verticales de la closure se refieren a argumentos que se recibirán en la invocación de la closure.

En el siguiente código, la closure utiliza una referencia a la variable *x*. Es un ejemplo de closure del tipo *Fn*:

```
fn main() {
   let x = 10;
   let cuadrado = || {x*x};
   println!("{}", cuadrado(x)); // Imprime 100
}
```

El siguiente ejemplo utiliza una referencia mutable a la variable *x* del contexto. Se trata de una closure del tipo *FnMut*. Es necesario declarar como mutables la variable *x* y la closure *incrementa_x*:

```
fn main() {
   let mut x = 10;
   let mut incrementa_x = || x+=1;
   incrementa_x();
   println!("{}", x); // Imprime 100
}
```

La siguiente closure, en cambio, solo se puede ejecutar una vez, pues la variable *v* se *consume*, esto es, la propiedad se pasa a una variable local de la closure. Es un ejemplo de closure del tipo *FnOnce*[4]:

```
fn main() {
   let v = vec![1, 2, 3];
   let consume_x =  || {let w = v;};
   consume_x();
   consume_x(); // ¡ERROR, v se ha movido fuera de su contexto!
}
```

Se puede usar la cláusula *move* para que la closure tome la propiedad del valor de una variable del contexto. Esto no impide que la closure se ejecute más de una vez, pero impide volver a utilizar dicha variable fuera de la closure:

```
fn main() {
   let x = vec![1, 2, 3];
   let captura_x =  move || println!("{:?}", x);
   captura_x();
   captura_x();
   println!("{:?}", x); // ERROR, x se ha movido a la closure
}
```

[4] El concepto de *propiedad* de los valores se explicará en el Capítulo 4. Es posible que algunos de los conceptos que se están explicando en relación con las closures sea necesario releerlos tras leer dicho capítulo, a fin de comprenderlos mejor.

Cuando se utiliza una variable del contexto dentro de una closure, se está compartiendo la propiedad de manera inmutable o de manera mutable. Dicha circunstancia estará activa mientras lo esté la closure y se regirá por las reglas generales sobre la forma de compartir referencias. Así, si se comparte una referencia mutable a un valor del contexto, no pueden existir simultáneamente otras referencias, ni mutables ni inmutables al mismo valor.

Observe el siguiente ejemplo:

```
fn main() {
   let mut x = vec![1, 2, 3];
   let mut modifica_x =  || {x.push(4); println!("{:?}", x)};

   modifica_x(); // Referencia mutable
   x.push(5);    // Referencia mutable
   println!("{:?}", x); // Referencia inmutable
}
```

El código anterior sí que compila, a pesar de que se utilizan dos referencias mutables (*modifica_x()* y *x_push(5)*) y una referencia inmutable (*printf!()*). Ninguna de las referencias compartidas se intenta utilizar por segunda vez después de haber usado otras. Así, si se hace una segunda llamada a *modifica_x()* después de haber utilizado la referencia inmutable o la otra referencia mutable, como se hace en el siguiente código, el compilador señalará dos errores: uso simultáneo de dos referencias mutables y uso simultáneo de una referencia inmutable mientras existe una referencia mutable:

```
fn main() {
   let mut x = vec![1, 2, 3];
   let mut modifica_x =  || {x.push(4); println!("{:?}", x)};
   modifica_x();
   x.push(5); // Segundo uso de una referencia mutable
   println!("{:?}", x); // Uso de una referencia inmutable
   modifica_x(); // Esta línea hace saltar los errores
}
```

Este segundo uso de *modifica_x()* hace ver al compilador que, entre los dos usos, la referencia mutable está activa y, por tanto, no permite la utilización entre

medias de otras referencias, ni mutables ni inmutables.

Observe el siguiente código:

```
fn main() {
   // Se declara y asigna x en el contexto
   let mut x = 10.9;
   // f accede al valor de x
   let f = || {return x};
   // Se modifica el valor de x en el contexto
   let x = 17.0;

   // f() sigue usando el antiguo valor de x
   println!("{:?}", f()); // Imprime 10.9
   // Dentro del contexto, x ha cambiado de valor
   println!("{v:?}");     // Imprime 17.0
}
```

Cuando una closure accede a una variable del contexto, lo hace al valor que tenía dicha variable en el momento de declarar la closure. Si con posterioridad a la declaración de la closure se modifica el valor de la variable, la closure seguirá utilizando el valor que tenía la variable cuando se declaró la closure. El código del ejemplo anterior muestra claramente esta circunstancia.

El uso de closures es un patrón habitual en Rust. Son útiles en diversas situaciones:

- Se pueden usar como argumentos de otras funciones.
- Se usan profusamente con los iteradores, en métodos como *map()*, *filter()* y otros.
- Se pueden usar como campos de una estructura y definir un comportamiento de la misma.

El uso como parámetros de funciones, ya sean funciones ordinarias o métodos de los iteradores, se verá más adelante. El código que sigue sería un ejemplo del tercer uso indicado. Se trata de una estructura que simboliza un *botón* que llama a una closure cuando se pulsa:

```
struct Button<F> {
   callback: F,
}

impl<F: Fn()> Button<F> {
   fn new(callback: F) -> Self {
      Self { callback }
   }
   fn click(&self) {
      (self.callback)();
   }
}

fn main() {
   let button = Button::new(|| println!("¡Botón pulsado!"));
   button.click();  // Imprime: ¡Botón pulsado!
}
```

3.15 *Traits*

La palabra inglesa *trait* se puede traducir por *rasgo*, *característica*, *atributo* o *cualidad*. Los *traits* de *Rust* son similares a los *interfaces* existentes en otros lenguajes. Los *traits* definen, de manera abstracta, las funcionalidades que deben implementar los tipos de datos adscritos al *trait*. También sirven para limitar qué tipos concretos forman parte de un genérico. Para no dar lugar a errores de interpretación, a lo largo del texto se utilizará la palabra *trait* en inglés.

Las librerías del lenguaje *Rust* incluyen muchos *traits* predefinidos. Cada tipo de datos definido en el lenguaje implementa algunos de esos *traits*. Por ejemplo, el *trait Iterator* define métodos como *next()* o *count()*. Los tipos de datos que implementan el *trait Iterator* implementan dichos métodos. En la librería estándar hay numerosos ejemplos de *traits* definidos en el lenguaje.

Definir un *trait* consiste en definir la signatura de una serie de métodos. La declaración y definición de un *trait* se hace con la palabra clave *trait*, seguida del nombre asignado a dicho *trait*. A continuación, entre llaves, se lista la signatura de los métodos que deberán implementar los tipos que se adhieran a dicho *trait*. La definición de un *trait* se hace de la siguiente manera:

```
trait NombreDelTrait {
   ... signatura de los métodos del trait ...
}
```

Por convenio, los *traits* se nombran en mayúsculas siguiendo el criterio *Camel Case*, con la primera letra de cada palabra en mayúsculas. Todos los tipos de datos que implementen el *trait* deberán tener definido el código de dichos métodos.

Para definir la implementación de un *trait* por parte de determinado tipo de datos, se utiliza la palabra clave *impl* seguida del nombre del *trait*, la palabra clave *for* y el nombre del tipo de datos:

```
impl NombreDelTrait for NombreDelTipoDeDatos {
   ... métodos del trait ...
}
```

Para ilustrar la forma de definir y utilizar los *traits*, se va a desarrollar un ejemplo que simula un termostato capaz de accionar varios dispositivos:

```
trait Regulable {
   fn activar(&self);
   fn desactivar(&self);
}
struct Termostato;
impl Termostato {
   fn activa_disp<T: Regulable>(&self, dispositivo: &T) {
      dispositivo.activar();
   }
   fn desactiva_disp<T: Regulable>(&self, dispositivo: &T) {
      dispositivo.desactivar();
   }
}
```

En primer lugar, se define el *trait Regulable* que tiene dos métodos llamados *activar()* y *desactivar()*. Cualquier dispositivo que tenga que ser accionado por el *Termostato* deberá implementar dicho *trait*. La estructura *Termostato* tiene unos métodos para activar y desactivar los dispositivos, que son los que utiliza para

manejar los aparatos. Estos métodos reciben como parámetro una referencia a un valor de un tipo genérico que implemente el *trait Regulable*. Esto garantiza que el dispositivo que reciban dichos métodos, implementará *activar()* y *desactivar()*, con lo que podrán ser manejados por el *Termostato*.

En el código se puede ver cómo se especifica que un tipo genérico debe implementar determinado *trait*: se ponen dos puntos tras la letra *T* del genérico y, a continuación, el nombre del *trait* que debe implementar; si tuviera que implementar más de un *trait*, se pone la lista de *traits* separados por el símbolo $+$. Cuando se limitan los tipos genéricos permitidos para un parámetro a aquellos tipos que implementan determinados *traits* se dice que se han impuesto *trait bounds* a los parámetros.

A continuación se muestra el código de dos de dichos dispositivos: *Radiador* y *Ventilador*:

```
struct Radiador;

impl Regulable for Radiador {
   fn activar(&self) {
      println!("Radiador encendido");
   }
   fn desactivar(&self) {
      println!("Radiador apagado");
   }
}

struct Ventilador;

impl Regulable for Ventilador {
   fn activar(&self) {
      println!("Ventilador en marcha");
   }
   fn desactivar(&self) {
      println!("Ventilador parado");
   }
}
```

Se puede observar cómo se ha implementado el código concreto de los métodos *activar()* y *desactivar()* para los tipos de datos *Radiador* y *Ventilador*: se

define un bloque *impl* que implementa los métodos *activar()* y *desactivar()* del *trait*, con las instrucciones concretas para el tipo de datos correspondiente.

Por último, se muestra la función *main()*, donde se crea un *Termostato* que acciona un *Radiador* y un *Ventilador*:

```
fn main() {
   let tato = Termostato;
   let radiador = Radiador;
   tato.activar_dispositivo(&radiador);
   let ventilador = Ventilador;
   tato.activar_dispositivo(&ventilador);
   tato.desactivar_dispositivo(&radiador);
   tato.desactivar_dispositivo(&ventilador);
}
```

Solo se puede implementar un *trait* en un tipo si el *trait* o el tipo son locales al código. Por ejemplo, se puede implementar el *trait Display* de la librería *std* en un tipo perteneciente al programa que se esté desarrollando. También se podría implementar dentro de un programa el *trait Regulable* para el tipo *Vec<T>* de la librería *std*. Lo que no se puede es implementar *traits* externos en tipos externos. Por ejemplo, no se puede implementar el *trait Display* en el tipo *Vec<T>*, pues los dos son externos al código del programa que se esté desarrollando.

3.15.1 Herencia en *traits*

Rust no es un lenguaje orientado a objetos, aunque tiene elementos que permiten simular algunas de las características de la programación orientada a objetos. No existen objetos como tales, pero las estructuras pueden tener datos y métodos asociados, con lo que pueden suplir algunas de las funcionalidades que se asignan a los objetos en otros lenguajes.

Rust no dispone del mecanismo de herencia entre tipos de datos, pero sí que permite implementar la herencia en los *traits*. Cuando se establece que un *trait B* es derivado a partir de otro *trait A*, los tipos de datos que implementen el *trait B* derivado, están obligados a implementar los métodos que establece el *trait A* del que deriva.

El siguiente ejemplo muestra el mecanismo de herencia de *traits*. Se define el *trait Mapa* que deriva del *trait Dibujable*:

```
trait Dibujable {
   fn dibuja(&self);
}
trait Mapa : Dibujable {
   fn bounding_box(&self) -> (i32, i32, i32, i32);
}
```

Los tipos que implementen el *trait Mapa* deberán implementar el método *dibuja()* del *trait Mapa* y el método *bounding_box()* del *trait Dibujable*.

3.15.2 Diferencias entre *traits* e *interfaces*

Rust no es un lenguaje orientado a objetos. Sin embargo, observando los códigos anteriores, se puede ver que la utilización de estructuras y enumeraciones combinadas con *traits* y genéricos permite realizar una programación similar a la programación orientada a objetos. No se dispone del mecanismo de herencia de tipos, pero sí que se dispone de un mecanismo de herencia de *traits*, que permite un tipo de programación más flexible, como se comentó en el Apartado 1.1.

Los traits de *Rust* son similares a los *interfaces* existentes en otros lenguajes, como en Java, por ejemplo. Hay algunas diferencias, no obstante, que conviene destacar:

- En *Rust* está definida la herencia entre *traits*, pero no existe ningún mecanismo de herencia entre tipos de datos. Esto quiere decir que se puede definir un *trait B* que deriva de un *trait A*, y los tipos que quieran implementar el *trait B* deberán implementar también el *trait A*. Pero no hay ningún mecanismo de herencia entre los tipos asociados con la implementación de uno u otro *trait*.

  En programación orientada a objetos, esto implica que los tipos de datos utilizan la composición, no la herencia. Esto no es malo; de hecho, en programación orientada a objetos suele ser recomendable utilizar la composición frente a la herencia.
- Como ya se ha indicado, se pueden crear *traits* para tipos externos al programa, aunque no se tenga acceso a los tipos.
- Los *traits*, en sí mismos, no pueden ser el valor devuelto por un método, como sucede con los *interfaces* en Java, por ejemplo. Para eso hay que devolver *trait objects*.

CAPÍTULO 4

Propiedad de los valores

Contenido

4.1 El concepto de propiedad de los valores
4.2 Referencias

Rust es un lenguaje de tipado estático, esto es, todos los tipos de datos de los valores asociados a cualquier variable tienen que ser conocidos en tiempo de compilación. El concepto de la propiedad de los valores es probablemente el elemento más disruptivo del lenguaje Rust en relación con otros lenguajes de programación.

En Rust, durante la ejecución de un programa, cada valor alojado en memoria tiene un propietario y se impone que sea único en cada momento. Se establecen reglas estrictas en relación con la forma de compartir dicha propiedad mediante referencias. Cuando la variable propietaria de un valor llega al final de su ámbito, se destruyen la variable y el valor asociado. Ninguna referencia puede sobrevivir al valor al que apunta.

El cumplimiento de las reglas de propiedad se realiza mediante un componente del compilador denominado "borrow checker". Es frecuente entre programadores de Rust, entre "rustaceans", decir que se está peleando con el "borrow checker", para indicar las dificultades que se encuentran a veces para conseguir una compilación sin errores relacionados con el incumplimiento de las reglas de propiedad.

En este capítulo se va tratar el tema de la propiedad de valores.

4.1 El concepto de propiedad de los valores

En programación, una *variable* es un identificador (una etiqueta) que hace referencia a un determinado *valor* almacenado en alguna posición de memoria. Es habitual referirse al valor utilizando el nombre de la variable, pero las *variables* y los *valores* a los que referencian son conceptos diferentes.

Las variables, según el tipo de datos que lleven asociado, se comportan de manera diferente en las asignaciones, cuando se pasan como argumentos de funciones y en otras situaciones que se irán comentando.

Las variables de tipos de datos primitivos, funcionan *por copia de valor*. Observe el siguiente código:

```
fn main() {
   let x: i32 = 10;
   let y: i32 = x;
   println!("{}", y); // Imprime 10
   println!("{}", x); // Imprime 10
}
```

En el código anterior, se crea una variable *x* del tipo *i32* y se le asigna el valor 10. En la memoria se guarda la variable *x* asociada al valor 10. A continuación, se crea la variable *y*, a la que se asigna el valor de *x*. Lo que sucede es que se guarda en memoria la variable *y* asociada con una copia del valor que tiene *x*.

Tras esto, en memoria hay dos variables, y dos valores: *x*, *y*, 10 y 10.

La Figura 4.1 esquematiza la situación final de la memoria tras las dos asignaciones. Si se imprimen los valores de *x* e *y*, como se hace en el programa, se imprimen los dos valores.

Name	Type	Value
x	i32	10
y	i32	10

Figura 4.1: Esquema de la memoria tras ejecutar dos asignaciones de números i32

Este mecanismo de copia del valor en la nueva variable sucede siempre que los tipos implicados sean tipos primitivos y también con arrays y con tuplas de tipos primitivos.

Observe, en cambio, el comportamiento cuando se hace un programa similar al anterior, pero en el que las variables son vectores:

```
fn main() {
   let v = vec![10, 20];
   let w = x;
   println!("{:?}", w); // Imprime 10
   println!("{:?}", v); // ERROR, x no existe
}
```

El programa anterior crea una variable *v* de tipo *Vec<i32>* y asigna valor a sus componentes. A continuación, crea una variable *w* y le asigna el valor de *v*. En asignaciones de este tipo, los vectores funcionan de manera diferente que los tipos primitivos (vea el esquema de la Figura 4.2).

Cuando se crea la variable *w* y se le asigna el valor de *v*, no se copia el valor de las componentes de *v* en la nueva variable, lo que se asocia con la variable *w* es la posición de memoria donde están guardadas las componentes del vector. En otros lenguajes, por ejemplo Java, a partir de ese momento habría dos variables en memoria, *v* e *w*, apuntando a la misma posición de memoria; se dice que Java *asigna por referencia*, no por valor. En Rust, el funcionamiento es diferente: la variable *w* se queda apuntando al valor, esto es, a la posición de memoria donde se guardan las componentes del vector, mientras que la variable

v queda inaccesible. Se dice que el valor de las componentes del vector se ha *movido* de la variable *v* a la variable *w* o que la *propiedad* del valor ha pasado de *v* a *w*.

La Figura 4.2 esquematiza este proceso. En la parte izquierda de la figura se muestra la situación tras la creación de la variable *v*: se guarda la variable *v* asociada a la posición de memoria en la que están almacenadas las componentes del vector, por ejemplo, la posición 5000. La parte derecha de la misma figura muestra la situación tras asignar a la variable *w* el valor de la variable *v*: la variable *v* deja de existir y la propiedad del valor pasa a la variable *w*[1].

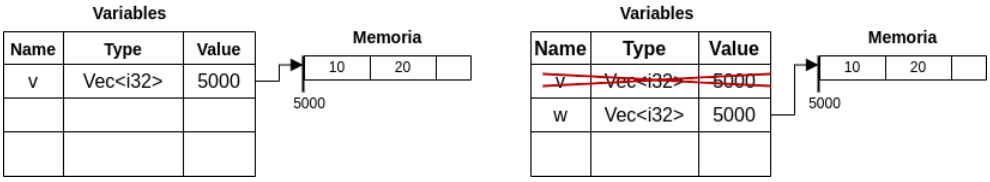

Figura 4.2: Izquierda: esquema de la memoria tras crear la variable *v* del tipo *Vec<i32>*. Derecha: esquema de la memoria tras asignar a la variable *w* el valor de la variable *v*

En programación, se entiende por *ámbito* de una variable (en inglés *scope*) la parte del programa en la que dicha variable es accesible. En Rust, el ámbito de una variable lo determina el bloque en el que se crea la variable. Los bloques están delimitados entre dos llaves. Por ejemplo, en el siguiente código, la variable *x* existe durante toda la extensión de la función *main()*; en cambio, la variable *y* solo existe dentro del bloque en el que se ha creado y cuando se intenta imprimir fuera de ese bloque, se produce un error.

```
fn main() {
   let x= 10;
   {
      let y = 20;
   }
   println!("{:?}", x); // Imprime 10
   println!("{:?}", y); // ERROR, y no existe
}
```

[1] En realidad, la forma de asignar la memoria de un vector en Rust es un poco más compleja de lo que se esquematiza en la Figura 4.2, pero a los efectos de la explicación del concepto de propiedad se ha considerado conveniente simplificarlo como se ha hecho.

Las reglas por las que se rige el concepto de la *propiedad de los valores* en Rust son las siguientes:

- Cada valor tiene un propietario.
- En todo momento, cada valor tiene un solo propietario.
- Si se termina el ámbito del propietario, se destruyen el valor y el propietario.

Los tipos de datos que no están afectados por el concepto de propiedad se dice que se rigen por una semántica *Copy*, mientras que los tipos que sí están afectados por el concepto de propiedad se dice que funcionan con semántica *Move*.

4.2 Referencias

Es posible tomar prestada la propiedad de un valor utilizando una referencia al mismo. El compilador de Rust se encarga de comprobar que ninguna referencia sobreviva más allá que el valor al que apunta.

Hay dos tipos de referencias:

- **Referencias inmutables (&):** permiten utilizar el valor al que apuntan, pero no permiten modificarlo. Pueden existir al mismo tiempo varias referencias inmutables a un mismo valor en memoria.
- **Referencias mutables (&mut):** permiten utilizar el valor al que apuntan y permiten también su modificación. Solo puede existir una referencia mutable a un valor en cada momento. Si durante la ejecución de un programa existe una referencia mutable a un valor en memoria, no pueden existir simultáneamente otras referencias al mismo valor, ni mutables ni inmutables.

Se denomina *desreferenciar* a obtener el valor al que apunta una referencia. En muchas situaciones, Rust desreferencia directamente. Por ejemplo, si ejecuta el siguiente código, el valor que se imprime en pantalla es el valor al que apunta la referencia, no la referencia en sí misma:

```
fn main() {
   let x = 32.6;
   let ref_x = &x;
   println!("{}", ref_x); // Imprime 32.6
}
```

Hay ocasiones, no obstante, en que es necesario forzar la desreferenciación explícitamente. Para ello, se usa el operador ∗ (asterisco) colocado delante de la referencia, como se hace en el siguiente ejemplo:

```
fn main() {
   let mut v = vec![10, 20, 30];
   for num in &mut v {
      if *num > 20 {
         *num = 40;
      }
   }
   println!("{:?}", v); // [10, 20, 40]
}
```

Si se quiere imprimir la dirección de memoria a la que apunta una referencia, hay que utilizar la especificación de formato *:p*:

```
fn main() {
   let x = 32.6;
   let ref_x = &x;
   println!("{}", ref_x); // Imprime 32.6
   println!("{:p}", ref_x); // Imprime 0x7fff189f4bb8
}
```

CAPÍTULO 5

Funciones de orden superior, closures e iteradores

En matemáticas y en ciencias de la computación, se denominan funciones de orden superior a las que cumplen al menos una de las siguientes condiciones:

- *Tomar una o más funciones como parámetro de entrada.*
- *Devolver una función como salida.*

En matemáticas, un ejemplo lo podría constituir la función derivada, que toma una función como parámetro y devuelve otra función.

En programación funcional, las funciones son consideradas elementos de primera clase, esto es, pueden usarse como argumentos de otras funciones, como valores devueltos y también asignarse a un símbolo o variable. La mayoría de los lenguajes han adoptado ya esta premisa y dan un tratamiento de primera clase a las funciones.

En este capítulo se va a hablar de funciones, de cómo asignarlas a variables y de cómo utilizarlas como argumento o como valor devuelto de otras funciones.

También se hablará de las closures, que son un tipo especial de funciones definidas dentro de otras funciones, que pueden acceder al contexto en el que se han definido.

Por último, se hablará de los iteradores y cómo encadenar procesos utilizando iteradores y funciones de orden superior.

5.1 Funciones de primera clase

En programación, se denominan *elementos de primera clase* a los elementos del lenguaje que se pueden asignar a variables y se pueden utilizar como argumentos o como valor devuelto por otras funciones.

No todos los elementos del lenguaje son elementos de primera clase. Por ejemplo, en muchos lenguajes, los operadores aritméticos como el $+$, el $-$ y otros, no son de primera clase. También sucede algo parecido con algunas construcciones, como los bucles o las bifurcaciones.

Es importante entender la diferencia entre el concepto de *declaración* y el de *expresión*. Un elemento de un lenguaje se considera que es una *declaración*

cuando no devuelve ningún valor; una *expresión*, en cambio, es un elemento que devuelve un valor.

En los lenguajes funcionales se da prioridad a las expresiones que devuelven valores. Así se hace en Rust. Algunas construcciones que en otros lenguajes son declaraciones, en Rust son expresiones. Es el caso de las bifurcaciones *if* o de los bucles *loop, for* o *while*, como se indicó en los apartados 3.12 y 3.13. En cambio, el operador *let* es una declaración que no devuelve ningún valor.

Se podría realizar un código como el siguiente:

```
fn main() {
   let x = 2;
   println!("{}", suma(if x==2 {3} else {4}, 5));

   let mut n = 1;
   println!("{}", loop {
      n=n+1;
      if n>3 {
         break 10
      }
   });
}
fn suma(x: i32, y: i32) -> i32 {
   x+y
}
```

En el programa anterior, se utiliza una bifurcación *if* como argumento de la función *suma()* y un bucle *loop* como argumento de la macro *println!()*. Observe que, en realidad, el argumento es el valor que devuelven dichas expresiones.

Actualmente, numerosos lenguajes se han adherido a la línea marcada por los lenguajes funcionales y consideran las funciones como elementos de primera clase, esto es, permiten que las funciones se puedan asignar a variables y se puedan utilizar como parámetros o como valor devuelto por otras funciones. En Rust, las funciones son objetos de primera clase.

Se denominan *funciones de primer orden* o también *funciones ordinarias* aquellas en las que ni los parámetros ni el valor devuelto son otras funciones. Por contra, las *funciones de orden superior* son aquellas en las que o bien alguno de los parámetros o bien el valor devuelto son una función.

Hay varias maneras de definir funciones en Rust:

- Dentro de un módulo o fichero *.rs*, con la visibilidad que le corresponda y acceso a otras funciones y a sus variables locales .
- Dentro de otra función o bloque de código, con visibilidad restringida a dicho bloque y acceso a otras funciones, pero no a variables del contexto del bloque en el que se ha definido.
- Cómo función asociada a un tipo de datos personalizado, como es el caso de las estructuras y las enumeraciones.
- Como closure definida dentro de otra función o bloque de código. Su visibilidad estará restringida al bloque en el que se ha definido, pero las closures sí que tienen acceso a las variables existentes en el contexto del bloque en el que se haya definido.

Ya se comentó este tema en el Apartado 3.14. También se habló allí de los tres tipos de funciones que considera Rust, en función del uso que hacen de la propiedad de los parámetros: el tipo *Fn*, para funciones ordinarias, el tipo *FnMut*, para funciones con parámetros mutables y el tipo *FnOnce* para funciones que solo se pueden invocar una vez.

5.1.1 Asignación de funciones a variables

Cualquiera de los tipos de funciones presentes en Rust se pueden asignar a una variable. Luego, se puede utilizar la variable como si fuera la función.

El siguiente código asigna a una variable de nombre *raiz2* la función *sqrt* de los números *f64*. Como resultado, la variable *raiz2* es del tipo *fn(f64) ->f64*, esto es, una función que recibe un argumento del tipo *f64* y devuelve un valor *f64*. Observe la forma de calcular la raíz cuadrada utilizando la variable así definida:

```
fn main() {
   let raiz2  = f64::sqrt;
   let y = raiz2(4.0); // Equivale a 4.0f64.sqrt()
   println!("{}", y);
}
```

Se podría asignar una función previamente codificada, como se hace en el siguiente código:

```
fn main() {
   let f = saludo;
   f(); // Imprime Hola, ¿qué tal?
}
fn saludo() {
   println!("Hola, ¿qué tal?");
}
```

En este caso, la variable *f* es del tipo *fn()*, esto es, del tipo de las funciones que no reciben argumentos y no devuelven ningún valor.

Las closures también se pueden asignar a variables, como se hace en el siguiente ejemplo:

```
fn main() {
   let cuadrado = |x| x*x;
   println!("{}", cuadrado(3.0)); // Imprime 9
}
```

Este tipo de funciones que no se definen dentro de un bloque *fn* con nombre, se denominan *funciones anónimas*. En este caso, la función anónima se ha asignado a la variable *cuadrado*.

También se suele decir que la closure se ha codificado *inline*. El término *inline* se refiere a que la codificación de la función se ha hecho dentro de la línea de código de otra instrucción. Es habitual también codificar closures *inline* cuando se utilizan como parámetros de otras funciones.

5.1.2 Utilización de funciones como parámetros de otras funciones

Observe el código siguiente:

```
fn main() {
   let pi = std::f64::consts::PI;
   println!("{}", operador(f64::sin, pi/2.0)); // Imprime 1
   println!("{}", operador(f64::cos, pi)); // Imprime -1
   println!("{}", operador(|x| x*x, 4.0)); // Imprime 16
   println!("{}", operador(duplica, 5.0)); // Imprime 10
}
fn operador(f:fn(f64) -> f64, x: f64) -> f64 {
   f(x)
}
fn duplica(x: f64) -> f64 {
   x*2.0
}
```

En el código anterior, la función *operador* es una función de orden superior que se ha definido con la siguiente signatura:

```
fn operador(f:fn(f64) ->f64, x: f64) ->f64
```

Cualquier función con un único parámetro del tipo *f64* y que devuelve un valor del tipo *f64*, se podrá utilizar como argumento en la llamada a la función *operador()*. La función *operador()* devolverá el resultado de aplicar la función *f* que recibe como primer argumento al valor *x* del tipo *f64* que recibe como segundo argumento.

En algunos lenguajes, a la función que se pasa como argumento, para que la función de orden superior la invoque, se le denomina *función de callback*.

5.1.3 Uso de funciones como valor devuelto por otras funciones

En el siguiente código se define una función de orden superior, llamada *operador*, que recibe un número entero como argumento. Si el valor recibido es mayor que cero, *operador()* devuelve la función *f64::sin*; en caso contrario, devuelve la función *f64::cos*.

```
fn main() {
   let pi = std::f64::consts::PI;
   let seno = operador(3);
   println!("{}", seno(pi/2.0)); // Imprime 1
   let coseno = operador(-2);
   println!("{}", coseno(pi)); // Imprime -1
}
fn operador(n: i32) -> fn(f64) -> f64 {
   if n>0 {f64::sin} else {f64::cos}
}
```

Otra forma de indicar el tipo de función que se pasa como parámetro o que se devuelve como resultado es utilizar la cláusula *impl* y el nombre del *Trait*, que en este caso es parecido pero poniendo *Fn* en mayúsculas:

```
fn operador(n: i32) -> impl Fn(f64) ->f64
```

5.2 Errores de ejecución

El mecanismo de excepciones es la forma habitual que ofrecen muchos lenguajes para gestionar los errores que se producen durante la ejecución de los programas. Pueden ser errores debidos a la entrada de un dato incorrecto, al intento de acceso a un fichero que no existe, el intento de acceder a una página web sin tener conexión u otros tipos de errores de ejecución. En muchas ocasiones, el error de ejecución proviene de la existencia de un puntero nulo en alguna parte del programa.

El invento del puntero nulo se atribuye a Tony Hoare, durante la implementación de los tipos de datos para el lenguaje ALGOL W, a mediados de la decada de 1960. Años después, él mismo lo llamó *el error de los mil millones de dólares*[1]:

> *"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."*

[1] Esta cita se ha tomado de la Wikipedia

En Rust no existe el puntero *null*. Tampoco existe un mecanismo de excepciones para el tratamiento de los errores de ejecución. En Rust se distingue entre dos tipos de errores de ejecución:

- *Errores recuperables*: por ejemplo, se pide al usuario el nombre del fichero que se pretende abrir y se comprueba que dicho fichero no existe. En esos casos, lo que se puede hacer es informar del error al usuario y volver a pedirle el nombre del fichero.
- *Errores no recuperables:* por ejemplo, se necesita acceder a un dispositivo y el dispositivo no existe. En estos casos se interrumpe la ejecución del programa, quizás imprimiendo un mensaje que informa al usuario del error que se ha producido.

En Rust, cuando el error da lugar a la interrupción del programa se suele decir que el programa *ha entrado en pánico*, por analogía con el nombre de la macro *panic!()* que se suele utilizar para forzar interrupciones.

La macro *panic!()* recibe un mensaje de texto como argumento, interrumpe el programa y muestra en la consola dicho mensaje junto con otras informaciones.

Puede probar el siguiente código:

```
fn main() {
   panic!("¡Programa interrumpido!")
}
```

Para gestionar los errores recuperables se suelen utilizar las enumeraciones *Option<T>* y *Result<T>*, en la forma que se va a explicar en los siguientes apartados.

5.3 El *enum Option*

La enumeración *Option* tiene la siguiente declaración:

```
pub enum Option<T> {
   None,
   Some(T),
}
```

El tipo *T* puede ser cualquiera. Es fácil definir un valor del tipo *Option*. Por ejemplo, el siguiente código declara dos variables del tipo *Option<u32>*, una con cierto valor y otra sin valor:

```
let n = Some(3u32);
let m: Option<u32> = None;
```

Observe que, en el caso de la variable *n* que sí tiene valor, el tipo de datos viene definido por el valor en sí, mientras que en el caso de la variable *m* hay que especificr el tipo de datos del *Option*.

El tipo *Option<T>* se utiliza para encapsular un valor que puede no existir. Procede de los lenguajes funcionales y del concepto de *mónada*. En estos lenguajes es habitual encontrar el tipo *Maybe* (“tal vez un”), representando un valor que es o un valor del tipo *T*, o ninguno.

Los métodos *is_some()* e *is_none()* permiten saber si determinada variables del tipo *Option* es la variante *Some* o la variante *None*:

```
fn main() {
   let n = Some(3);
   let m: Option<u32> = None;
   assert!(n.is_some());
   assert!(m.is_none());
}
```

Cuando la variante de la variable es *Some*, se puede extraer su valor con el método *unwrap()*:

```
let n = Some(3);
assert_eq!(n.unwrap(), 3);
```

La función *unwrap()* interrumpe la ejecución del programa con *panic!*, si la variante es *None*. Por ello, se suele utilizar más el método *unwrap_or()*, que proporciona un valor del mismo tipo que el contenido en el *Option* como valor devuelto en el caso de que la variante sea *None*:

```
assert_eq!(Some("car").unwrap_or("bike"), "car");
assert_eq!(None.unwrap_or("bike"), "bike");
```

También existen unos métodos *unwrap_or_else()* y *unwrap_or_default()*, que puede consultar en la documentación del tipo *Option*.

Es habitual utilizar el tipo *Option* como resultados de funciones. Suponga una función que reciba dos parámetros enteros y devuelva la división de uno entre el otro. En Java, la solución podría ser la siguiente:

```
static int dividir(int num, int den) {
   return num/den;
}
```

En Java, si el denominador es cero, se lanzará una excepción y se interrumpirá el programa. En principio hay dos posibilidades para gestionar una función de ese tipo: capturar la excepción o interceptar los denominadores cero. [2].

Como se ha dicho, Rust no dispone de tipo *null* ni de ningún mecanismo de excepciones para la gestión de errores. En Rust, en cambio, se puede imponer que la función *dividir()* devuelva un *Option<i32>* en lugar de un valor entero. El código podría ser el siguiente:

```
fn main() {
   let n = dividir(3, 0);
   assert_eq!(n, None);
}
fn dividir(num: i32, den: i32) -> Option<i32> {
   match den {
      0 => None,
      _ => Some(num/den)
   }
}
```

Una ventaja de que la función devuelva un *Option* es que el programa está obligado a gestionar la posibilidad de que el valor devuelto no exista. En el caso

[2]No se podría hacer que la función devuelva *Infinity* pues no existe para valores del tipo *int*.

del programa Java, nada indica en la signatura de la función ni en el cuerpo de la misma que cabe la posibilidad de que se genere una excepción. Ello podría dar lugar a errores de ejecución no previstos. En el caso resuelto con Rust, el programa que llama a la función sabe que puede obtener como resultado la variante *None* y el compilador le obliga a gestionar los dos posibles resultados[3].

La ventaja de devolver resultados de funciones con *Option* es más evidente cuando hay que encadenar varias funciones en las que una o más de ellas pueden devolver resultados no válidos. En el Apartado 13.6 se mostrarán ejemplos de encadenamiento de funciones con iteradores.

5.4 Gestión de errores con *Result*

Como se ha mencionado, en *Rust* no existe el mecanismo de excepciones que se utiliza en otros lenguajes para resolver las situaciones en las que una operación pueda dar lugar a un error de ejecución. Tampoco existe un valor *null*. El módulo *result* de la librería *std* de *Rust* proporciona la enumeración *std::result::Result* para gestionar dichas situaciones. Cuando una operación pueda dar lugar a un resultado correcto o erróneo, lo que se hace es devolver una instancia de *Result*.

La enumeración *std::result::Result* se define de la siguiente manera:

```
pub enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

Las dos variantes de esta enumeración son:

- *Ok(T)*: representa el éxito de una operación y proporciona un valor del tipo de datos genérico *T*.
- *Err(E)*: representa que se ha producido algún error en la operación y proporciona un valor del tipo de datos genérico *E*.

Se va a desarrollar a continuación una función para dividir dos números del tipo *f64*. Para resolver la situación de que el divisor sea cero, la función devuelve un valor de la enumeración *Result* que se acaba de describir:

[3] Java ofrece también un tipo *Optional<T>* que, junto a otros tipos, permite acometer éste y otros aspectos de la programación funcional.

```
fn div(x: i32, y: i32) -> Result<i32, ()> {
   match y {
      0 => Err(()),
      _ => Ok(x / y)
   }
}
fn main() {
   let d = div(10, 0);
   match d {
      Ok(value) => println!("{}", value),
      Err(()) => println!("Error, denominador cero")
   }
}
```

Como se verá en el apartado 13.5, otra forma de gestionar los errores devueltos en un *Result* es utilizar el operador *?* para propagarlos hacia arriba, hacia la función que llama.

Un problema asociado a los lenguajes que utilizan el mecanismo de excepciones es que se pueden capturar las excepciones e ignorarlas. El tipo *Result* está definido con la anotación *#[must_use]*, que indica al compilador que se emita un mensaje de aviso si se ignora el resultado de una función que devuelva un *Result*. El lector puede probar a sustituir el código de la función *main()* del ejemplo anterior por el siguiente:

```
fn main() {
   div(10.0, 0.0);
}
```

Si ahora trata de compilar el programa, el compilador emitirá un aviso de que el *Result* que devuelve la función no se ha utilizado.

El lector puede comprobar también que, si el resultado de la función *div()* se asigna a una variable, no se genera el aviso *must_use*. Pruebe a compilar el siguiente código:

```
fn main() {
   let _x = div(10.0, 0.0);
}
```

En el código anterior, el nombre de la variable se ha iniciado con un guion bajo. Si no se hace así, el compilador genera otro aviso, el de que la variable *x* no se utiliza. Es una demostración más de las numerosas comprobaciones que hace el compilador de *Rust*.

5.5 El operador *?*

Una manera habitual de tratar el objeto *Result* que devuelven algunos métodos es mediante una construcción *match*, como se ha visto en el apartado anterior. Dentro de una función que devuelva un *Result* o un *Option*, se puede sustituir dicha construcción por el operador *?*, escrito a continuación de la función que devuelve el *Result*. Con el operador *?*, si el resultado es *Ok*, el operador *?* hace el *unwrap()*; si el resultado es *Err*, se vuelve de la función en la que esté, devolviendo el error. De esta manera, se deja la gestión del error para la función que llamó. Si se está dentro de *main()*, se devuelve al sistema operativo. Este mecanismo se denomina *propagación de los errores hacia arriba*. El siguiente ejemplo utiliza el operador *?*:

```
fn main() -> Result<(), ()> {
   let _d = div(10.0, 0.0)?;
   Ok(())
}
```

Como se ve en el código anterior, ahora la función *main()* tiene que devolver un *Result*. Si se ejecuta el programa anterior, la división genera un error, que se muestra en el sistema operativo como *Error: ()*. En el ejemplo, si no hay error, se devuelve la variante *Ok()* de *Result*.

5.6 Iteradores

Un *Iterador* es un objeto que permite recorrer ordenadamente los elementos de una fuente de datos. En muchas ocasiones, la fuente de datos es una colección de objetos, pero también se pueden utilizar iteradores para leer ficheros línea a línea o para leer streams procedentes de un puerto de comunicaciones, por ejemplo.

Cuando un tipo de datos implementa el *trait Iterator*, se convierte en una especie de colección, que es posible recorrer de manera ordenada. Por ello, es habitual referirse a los elementos de dicho tipo de datos como *los elementos del iterador*, como si se tratase de una colección.

El uso de iteradores es un patrón de diseño muy utilizado en la programación funcional y que se ha incorporado progresivamente en todos los lenguajes de programación. Su origen se marca en el lenguaje CLU del año 1973 [10].

Técnicamente, un iterador en Rust es un tipo de datos que implementa el *trait Iterator*, que impone la existencia del método *next()*. Este método, si hay más elementos por iterar, devuelve *Some(E)*, donde *E* es el tipo de datos del iterador; si no quedan más elementos por iterar, devuelve *None*.

Las colecciones que ofrece el lenguaje Rust —arrays, vectores, hashmaps y otras— permiten obtener un iterador sobre los elementos de la colección mediante el método *into_iter()*, o bien un iterador sobre referencias a los elementos de la colección utilizando el método *iter()*. La diferencia fundamental entre ambos métodos es que, si se utiliza el método *into_iter()*, se *consume* la colección, pasando la propiedad al proceso que la esté utilizando; el método *iter()*, en cambio, utiliza referencias y no *consume* la propiedad de la colección.

El siguiente ejemplo utiliza el método *iter()* con un vector. Tras recorrer el vector, los elementos de la colección siguen estando disponibles:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];
   for num in v.iter() {
      println!("{}", num);
   }
   println!("{}", v[0]);
}
```

En cambio, si en el ejemplo anterior se utiliza el método *into_iter()*, se consume el vector en el bucle y los elementos de la colección ya no estarán accesibles después de la ejecución del mismo:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];
   for num in v.into_iter() { // into_iter() mueve la propiedad
      println!("{}", num);
   }
   println!("{}", v[0]); // Error, v se movió al bucle
}
```

Es posible utilizar un bucle *for* para recorrer una colección sin llamar explícitamente a ninguno de los dos métodos. El problema será similar: si se recorre la colección, se pasa la propiedad al bucle, mientras que si se recorre utilizando una referencia, la colección no se consume en el bucle.

El siguiente ejemplo utiliza una referencia a la colección y los elementos del vector siguen disponibles tras finalizar el bucle:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];

   for num in &v { // Se utiliza una referencia &v
      println!("{}", num);
   }
   println!("{}", v[0]); // v sigue disponible
}
```

En cambio, en el siguiente ejemplo, el bucle utiliza directamente la colección y consume la propiedad de la misma. Al finalizar el bucle, no se puede acceder a los elementos de la colección:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];
   // Se recorre la colección, se pasa la la propiedad de v
   for num in v {
      println!("{}", num);
   }
   // Ahora v ya no está disponible
   println!("{}", v[0]); // Error
}
```

En Rust, al definir los iteradores, se hace referencia al tipo de elementos que recorre, pero no al tipo de colección o a la fuente de los datos que los alberga. Ésto permite definir funciones que describen la intención de lo que se quiere conseguir al procesar una fuente de datos, de manera independiente a la forma que adopten dichos datos. De esta forma, se podría cambiar la fuente de datos o su forma interna de organización, manteniendo las funciones que procesan dichos datos. Se trata de una codificación más declarativa que los bucles.

Los iteradores proporcionan numerosos métodos que permiten operar con los elementos que se iteran. Básicamente hay dos tipos de métodos:

- Métodos que devuelven un nuevo iterador obtenido al realizar alguna operación sobre los elementos del iterador original. Se suelen llamar *adaptadores* o también *iteradores internos*. Ejemplos de estos métodos serían *map()* y *filter()*.
- Métodos que realizan alguna operación directamente sobre los elementos del iterador. Se suelen denominar *iteradores externos*. Un ejemplo de estos métodos sería *for_each()*.

Los métodos adaptadores permiten encadenar operaciones haciendo que la salida de cada uno ellos, sea la entrada del siguiente. Ésto favorece la *canalización del procesamiento* de los datos (*processing pipeline*). La codificación obtenida es declarativa y muestra de manera explícita qué se quiere hacer con los datos, en contra de lo que sucede al realizar el procesamiento a base de bucles.

La totalidad de los métodos que proporcionan los iteradores se pueden consultar en la documentación de la librería estándar, en la siguiente dirección:

`https://doc.rust-lang.org/std/iter/trait.Iterator.html`

Una característica de los iteradores en Rust es que actúan de manera perezo-

sa, esto es, las operaciones asociadas a sus métodos no se ejecutan hasta que son necesarias. En concreto, es habitual tras una serie de operaciones encadenadas utilizando los métodos del iterador, querer convertir el iterador resultante en algún tipo de colección, como un vector u otra. Para ello, se suele utilizar el método *collect()*, que permite obtener una colección a partir de un iterador. En ese momento se ejecutarán las operaciones. En los ejemplos que siguen se verá cómo utilizar el método *collect()* en diferentes situaciones.

Muchos de los métodos que se utilizan cuando se trabaja con iteradores son más o menos estándar y están disponibles en todos los lenguajes que hacen uso de iteradores, si bien, en algunos casos pueden utilizar nombres diferentes.

Los métodos de los iteradores es frecuente que sean ejemplos paradigmáticos de funciones de orden superior: reciben una función como argumento que es la que se utilizará para operar sobre los elementos del iterador. También es frecuente que la función que se pasa como argumento sea una closure.

Tres de estos métodos que podríamos decir que son estándar en cualquier lenguaje que utilice iteradores son los métodos *map*, *filter* y *fold*:

- *map()*: este método recibe como argumento una función que se aplica sobre cada uno de los elementos del iterador. El método *map* devuelve un nuevo iterador cuyos elementos no tienen por qué ser del mismo tipo que los del iterador original.
- *filter()*: recibe como argumento una función que al aplicarla a los elementos del iterador devuelve *true* o *false*. A esta función que recibe como argumento se le llama el *filtro*. El método *filter* devuelve un nuevo iterador con los elementos del iterador original en los que el filtro devuelve *true*.
- *fold()*: este método acumula los valores del iterador en un valor que puede ser de otro tipo diferente que el de los elementos del iterador. También utiliza una función como argumento. Es posible encontrar este método en otros lenguajes con el nombre *reduce*. En Rust también hay un método *reduce*, similar a *fold()*.

En los siguientes apartados se van a tratar en profundidad estos tres métodos.

5.7 El método *map()*

Como se ha dicho, el método *map()* recibe como argumento una función y la aplica a los elementos del iterador para obtener un nuevo iterador. La función que se pasa como argumento a *map()*, toma como argumento un elemento del

iterador original y devuelve otro elemento, que puede ser del mismo tipo o de otro. El resultado final es un nuevo iterador con elementos que no tienen por qué ser del mismo tipo que los del iterador original. La Figura 13.1 muestra el esquema de funcionamiento de *map()*.

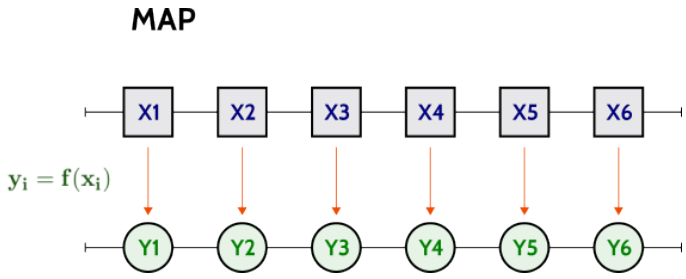

Figura 5.1: Esquema de funcionamiento de *map()*

En el siguiente código se crea un vector *v* de números enteros y se obtiene un nuevo vector formado por los cuadrados de dichos números.

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];

   let iterador_original = v.into_iter();
   let new_iterador = iterador_original.map(|x| x*x);
   let new_vector: Vec<i32> = new_iterador.collect();

   println!("{:?}", new_vector); // Imprime [1, 4, 9, 16, 25]
}
```

Observe los pasos que se han seguido:

1. Se obtiene un iterador a partir del vector *v* utilizando el método *into_iter()*.
2. Se obtiene un nuevo iterador aplicando el método *map* al iterador original. El método *map* recibe como argumento una closure que, para cada valor *x* del iterador original, devuelve su cuadrado.
3. El iterador resultante de la aplicación de *map()* se convierte de nuevo en un vector utilizando el método *collect()*. Al método *collect* hay que indicarle el tipo de colección que se quiere obtener, que en este caso es *Vec<i32>*.

Aunque en el ejemplo se ha desglosado cada paso para hacerlo más didáctico, lo habitual es hacer todas las operaciones encadenadas. También es habitual

ir poniendo las sucesivas operaciones en diferentes líneas, utilizando la tabuladores para dar claridad al código. El siguiente ejemplo es equivalente al anterior:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];

   let new_vector: Vec<i32> = v.into_iter()
      .map(|x| x*x)
      .collect();

   println!("{:?}", new_vector); // Imprime [1, 4, 9, 16, 25]
}
```

Como se ha dicho, el iterador resultante de la aplicación de *map()* puede tener elementos de distinto tipo que el iterador original. En el siguiente ejemplo, el iterador original es de cadenas de caracteres y se obtiene un iterador con el número de caracteres de cada una de esas cadenas:

```
fn main() {
   let v = vec!["Uno", "Dos", "Tres", "Cuatro", "Cinco"];

   let new_vector: Vec<i32> = v.into_iter()
      .map(|x| x.len() as i32)
      .collect();

   println!("{:?}", new_vector); // Imprime [3, 3, 4, 6, 5]
}
```

5.8 El método *filter()*

El método *filter()* recibe como argumento un función de retorno lógico: *true* o *false*. El parámetro de la función es del tipo de los elementos del iterador. El resultado es un nuevo iterador formado por los elementos del iterador original que *pasan el filtro*, esto es, los elementos en los que la función filtro devuelve *true*. La Figura 13.2 esquematiza el funcionamiento de *filter()*.

El siguiente ejemplo parte de un vector de tuplas con las de dos coordenadas *f64* de una serie de puntos. Se aplica un filtro para extraer los puntos del vector

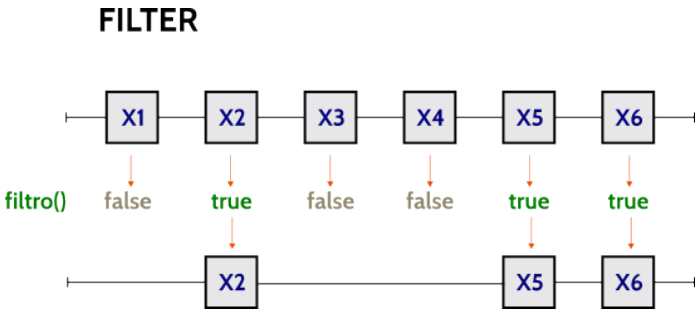

Figura 5.2: Esquema de funcionamiento de la función *filter()*

original que son interiores a un círculo de radio *1* centrado en el origen.

```
fn main() {
   let v: Vec<(f64, f64)> = vec![
   (1., 1.), (0.5, -0.25),
   (2., 0.1), (-0.5, -0.5)
   ];

   let new_vector: Vec<(f64, f64)> = v.into_iter()
      .filter(|(x, y)| (x * x + y * y).sqrt() < 1.0)
      .collect();

   println!("{:?}", new_vector); // [(0.5, -0.25), (-0.5, -0.5)]
}
```

Observe que en la función *filter*, a diferencia de lo que ocurría con la función *map()*, los elementos del iterador resultante son del mismo tipo que los del original. De hecho, son un subconjunto de los elementos del iterador original.

5.9 El método *fold()*

El método *fold()* aplica la función que recibe como argumento a los valores del iterador original, acumulando los valores en un solo valor que no tiene por qué ser del mismo tipo. A diferencia de *map* y *filter*, el método *fold()* tiene dos parámetros: el valor inicial del acumulador y la función que se utilizará para realizar los cálculos. La signatura de *fold()* es la siguiente:

```
fn fold(valor_inicial_acumulador: B, f)->B)
```

La función *f* es del tipo *FnMut*, recibe dos parámetros: el elemento del iterador

original que se está procesando y un valor del tipo de datos *B* del acumulado; la función debe devolver un valor del tipo de datos *B* del acumulado. Llamando *A* al tipo de datos de los valores del iterador original y *B* al tipo de datos del valor acumulado, esto se puede expresar así:

```
f: FnMut(B, T) ->B
```

Seguramente, un ejemplo ayude a comprender el funcionamiento de la función *fold()* mejor que tanta explicación. El siguiente código, calcula la suma de los elementos de un vector de números enteros:

```
fn main() {
   let v: Vec<i32> = vec![1, 2, 3, 4, 5];

   let suma: i32 = v.iter()
      .fold(0, |acc, x| acc+x);
   println!("{suma}"); // Imprime 15
}
```

Observe que, en este caso, el valor inicial del acumulador es *0*. La función acumuladora declara dos parámetros: *acc* y *x*. En cada iteración, *acc* contendrá el actual valor del acumulador. La función devuelve la suma de *acc* y *x*.

Observe también que, para obtener el iterador del vector, se ha utilizado el método *iter()*, por lo que las *x* de la función acumuladora son referencias a los valores del vector original y no se consume el vector.

La variable *acc* que se utiliza para el acumulador necesita ser mutable, de ahí que se haya dicho que la función acumuladora es del tipo *FnMut*.

En el siguiente ejemplo, los valores del iterador original son del tipo *char*, mientras que el valor acumulado es del tipo *String*:

```
fn main() {
   let v: Vec<char> = vec!['h', 'o', 'l', 'a'];

   let suma: String = v.into_iter()
      .fold(String::new(), |mut acc, c| {
            acc.push(c);
            acc
         }
      );
   println!("{suma}"); // Imprime hola
}
```

En este caso, ha sido necesario declarar que el parámetro *acc* de la closure es mutable. También ha sido necesario operar en dos pasos, pues la función *push()* modifica la cadena original pero devuelve el valor unitario *()*.

Otro ejemplo clásico es el cálculo del máximo de una serie de números:

```
fn main() {
   let v: Vec<i32> = vec![-1, 2, 5, 1, -2];

   let maximo: i32 = v.iter()
      .fold(v[0], |acc, x | if *x>acc {*x} else {acc});
   println!("{}", maximo); // Imprime 5
}
```

Un ejemplo un poco más complejo podría ser el calculo de la longitud de una polilínea definida mediante las coordenadas de sus puntos:

```
fn main() {
   let v: Vec<(f64, f64)> = vec![(0., 0.), (1., 1.), (2., 2.)];

   let acumulado: (f64, f64, f64) = v.iter()
      .fold((v[0].0, v[0].1, 0.0), |acc, p |{
         let d = ((p.0 - acc.0)*(p.0 - acc.0) +
                  (p.1 -acc.1)*(p.1 -acc.1)).sqrt();
         (p.0, p.1, acc.2 + d)
      }
   );
   println!("{:.4}", acumulado.2); // Imprime 2.8284
}
```

Observe que, en este caso, cada iteración necesita las coordenadas del punto anterior, de ahí la técnica que se ha utilizado para definir los valores acumulados como una tupla de tres componentes.

5.9.1 Ejemplo: cálculo del máximo

5.10 Herramientas de los Iteradores

Rust ofrece otras funciones que se pueden aplicar a los iteradores.

- **for_each():** devuelve la lista resultante de aplicar la closure que recibe como argumento a cada elemento de la lista original. Es equivalente a un bucle *for*, aunque en determinadas situaciones puede ser más rápido.
- **filter():** devuelve la lista resultante de filtrar los elementos de la lista original utilizando para ello la closure que recibe como argumento. La closure acepta como parámetro un elemento del tipo de datos de los elementos de la lista y devuelve *true* o *false*. La lista resultante contendrá solo los elementos en los que la closure devuelva *true*.

  El siguiente ejemplo extrae los elementos pares del array original:

```
fn main() {
   let lista = [1, 2, 3, 4, 5];
   let nueva_lista: Vec<&i32> = lista.iter().filter(|&x| *x%2==0).
   println!("{:?}", nueva_lista); // [2, 4]
}
```

En el ejemplo anterior, la función *filter()* opera sobre un iterador de referencias a valores, por lo que primero hay que convertir el array original en un iterador de referencias a números enteros usando el método *iter()*. La closure recibe una referencia a un elemento como parámetro, *&x*, y comprueba si el resto de dividir por 2 el valor al que apunta dicha referencia es igual a 0. Sucede que, a priori, no se conoce el tamaño del array resultante y el tamaño de un array hay que conocerlo en tiempo de compilación, por lo que se utiliza el método *collect()* para convertir el iterador resultante en un vector de referencias a números enteros.

La totalidad de las funciones aplicables a los iteradores se pueden consultar en el siguiente enlace:

(Fuentes:

- `https://blog.jetbrains.com/rust/2024/03/12/rust-iterators-beyond-the`
- Module iter: `https://doc.rust-lang.org/std/iter/`
- Processing a Series of Items with Iterators: `https://doc.rust-lang.org/book/ch13-02-iterators.html`
- `https://mustafabugraavci.blog/2024/04/23/mastering-iterators-in-rust`

CAPÍTULO 6

Funciones puras, inmutabilidad

Contenido

6.1 Distinguir entre datos, cálculos y acciones
6.2 Tipos de entradas y salidas de las funciones
6.3 Separar las cosas
6.4 Extraer los cálculos de las acciones
6.5 Ejemplo: eliminar entradas implícitas
6.6 Inmutabilidad. Copy-on-write

Como se indicó en el Capítulo 1, una de las características distintivas de la Programación Funcional es su preferencia por la utilización de las funciones puras, esto es, funciones sin efectos secundarios. Algunos autores utilizan la denominación cálculos, para referirse a las funciones puras y acciones, para referirse a las funciones con efectos secundarios [11]. Esta será la denominación que se utilizará preferentemente en el texto a partir de este momento.

Otro de los pilares de la programación funcional es la utilización de las estructuras de datos inmutables, lo que proporciona ventajas a la hora de hacer determinadas optimizaciones en el código compilado y facilita la programación concurrente y en paralelo.

En este capítulo se va a ahondar en la utilización de las funciones puras y variables inmutables. Las técnicas que se van a explicar, que proceden en su mayoría de patrones de diseño de la Programación Funcional, no dependen del lenguaje de programación que se utilice y su aplicación produce beneficios al código generado, haciéndolo más fácil de comprender, favoreciendo su reutilización y facilitando el proceso de pruebas y depuración. Se pueden aplicar por igual a la programación orientada a objetos o a la programación basada en procedimientos. En realidad, se podría decir que se trata simplemente de buenas prácticas de programación.

Hay dos ideas principales que subyacen en todas estas técnicas:

- *Separar las acciones de los cálculos y de los datos.*
- *Propiciar el uso de abstracciones de orden superior.*

6.1 Distinguir entre datos, cálculos y acciones

El código de los programas se puede dividir en tres categorías:

- Datos.
- Cálculos (funciones puras, funciones sin efectos secundarios).
- Acciones (funciones impuras, funciones con efectos secundarios).

Los *datos* son la plasmación de hechos que han sucedido. Los datos no producen efectos secundarios ni dan lugar a ningún resultado. Son código inerte. Lo que sí admiten, son diferentes interpretaciones. Un mismo dato, por ejemplo una cadena de texto con una palabra, puede admitir diferentes interpretaciones según el programa concreto que haga uso de ella.

Las *funciones puras* son aquellas que no producen efectos secundarios, no dependen de ningún estado exterior y siempre proporcionan el mismo resultado para el mismo valor de los argumentos de entrada (ver Figura 14.1). Se suelen denominar también *funciones matemáticas* o *cálculos*. En este texto, siguiendo la denominación utilizada por Eric Normand, se denominarán frecuentemente *cálculos* [11].

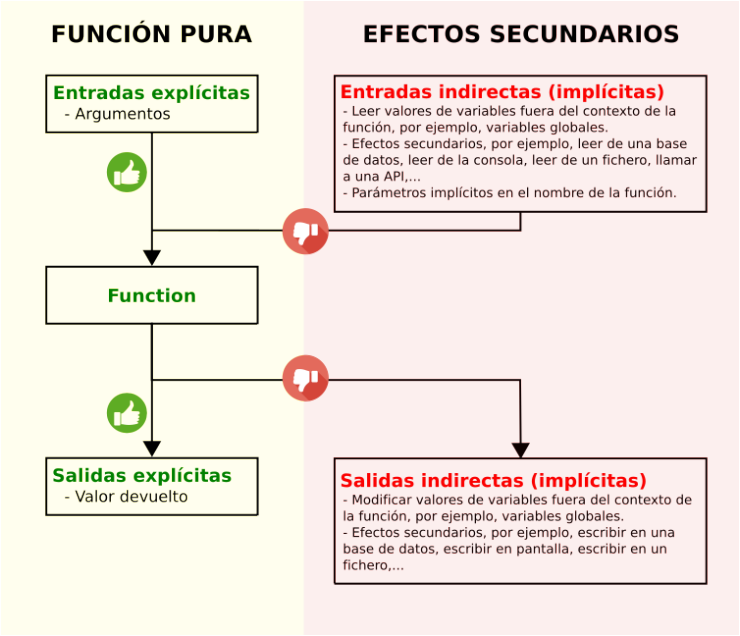

Figura 6.1: Funciones puras y efectos secundarios.

Se entiende por *efecto secundario* cualquier acción del programa que no sea un cálculo. Por ejemplo, modificar el estado de una variable, mostrar un mensaje en una pantalla o enviar un correo electrónico serían ejemplos de efectos secundarios. Cuando una función produce algún efecto secundario, se dice que es una *función impura*, una *función con efectos secundarios* o, siguiendo la denominación de Eric Normand, una *acción*. Así se denominarán frecuentemente en el texto a partir de ahora.

No es posible hacer un programa de utilidad sin producir efectos secundarios. De hecho, los efectos secundarios suelen ser la razón de ser de los programas. Lo que sí se puede hacer es codificar los programas de forma que las acciones siempre se lleven a cabo dentro de ámbitos controlados y bajo una supervisión minuciosa.

Al inspeccionar cualquier fragmento de código, hay que distinguir las partes que son acciones, de las que son cálculos y de las que son simplemente datos. En general, se debe dar preferencia a la utilización de datos sobre los cálculos y a los cálculos sobre las acciones.

Un ejemplo de cálculo sería la función siguiente:

```
fn cuadrado(x: f64) -> f64 {
   x*x
}
```

La función anterior, recibe como argumento un número del tipo *f64* y devuelve el cuadrado de dicho número. El código dentro de la función no afecta a nada externo a la misma, no tiene ninguna entrada de información que no sea a través de su parámetro ni ninguna salida que no sea a través de su valor devuelto; el valor que devuelve, siempre será el mismo si recibe el mismo valor como argumento, independientemente de en qué momento se llame a la función o cuántas veces se llame a la misma.

Las acciones, en cambio, son funciones con efectos secundarios. Los efectos secundarios pueden tomar distintas formas dentro del código. Por ejemplo:

- Llamadas a otras funciones o métodos: `println!("Hola, qué tal")`
- Constructores: `new Date()`
- Expresiones: `x` (si `x` es mutable).
- Declaraciones: `x = 3` (si se muta `x`).

Las funciones que son acciones, pueden estar compuestas por otras acciones, cálculos y datos. Una función que sea un cálculo puede estar compuesta por varios cálculos menores. Los datos solo pueden contener otros datos.

A continuación, se van a explicar con más detalle las características de los datos, los cálculos y las acciones.

6.1.1 Datos

Los datos son hechos resultantes de acontecimientos que han sucedido a lo largo de la ejecución del programa. Sus características principales son las siguientes:

- Se implementan utilizando los tipos de datos que ofrezca el lenguaje de programación que se esté utilizando.
- Las estructura de los datos encierra parte de su significado, por ejemplo, el orden de los elementos en una lista.
- Los datos en sí mismos, son inmutables. Los datos de un acontecimiento no varían, lo que puede suceder es que se produzcan nuevos acontecimientos que den lugar a nuevos datos.

- *Ventajas de los datos:* son serializables, se pueden comparar para comprobar su igualdad, están abiertos a distintas interpretaciones.
- *Desventajas:* el hecho de que admitan distintas interpretaciones puede ser un arma de doble filo. Un cálculo es útil, aunque no entendamos cómo está hecho, pero un dato no tiene ningún significado sin una interpretación adecuada.

En los programas, conviene separar la generación de los datos de su utilización. Por ejemplo, si un programa tiene que enviar unos correos a una serie de personas, conviene proceder primero a generar los correos, guardarlos en una estructura de datos y luego enviarlos.

Es preferible hacerlo así, que ir generando los correos uno a uno e ir enviando cada correo que se genera. Esto, facilitará la realización de test: es más fácil probar una función que genera una lista de datos que una función que envía un correo.

Si la lista de correos fuera muy grande, siempre se podría repetir el procedimiento anterior dividiendo en subgrupos los correos que se tienen que enviar.

6.1.2 Cálculos: funciones puras

Los cálculos son las funciones puras, también llamadas funciones matemáticas. A partir de una entrada se genera una salida. En una función que sea un cálculo, la entrada se produce a través de los parámetros de la función, la salida es el valor que devuelve y el cálculo es el cuerpo de la función.

Los cálculos gozan de *trasparencia referencial*, esto es, en cualquier lugar del código en el que aparece la llamada a una función que es un cálculo, se puede sustituir dicha llamada por el resultado que se obtiene. Cuando el compilador identifica un cálculo, puede proceder a optimizaciones sustituyendo la llamada a la función por su resultado. Esto no es posible garantizarlo en el caso de que la función sea una acción.

Las características de los cálculos:

- En los programas, los cálculos se implementan utilizando funciones.
- El significado que encierran es el cálculo que hacen.
- Las ventajas de los cálculos frente a las acciones son:
    - Es más fácil hacer test y probarlos.
    - Los test se pueden automatizar más fácilmente.
    - Es posible componer unos cálculos con otros.
    - No hay que preocuparse de otros cálculos que se pudieran estar pro-

duciendo al mismo tiempo, el resultado solo depende del valor de los argumentos de entrada (programación en paralelo).
    - Por el mismo motivo, no hay que preocuparse de lo que haya sucedido antes de llamar al cálculo o de lo que pueda suceder después.
    - No hay que preocuparse de cuántas veces se haya llamado anteriormente al cálculo.
- *Desventajas:* al igual que sucede con las acciones, no se puede estar seguro de lo que hacen si no se ejecuta el código. Lo único que se está seguro de lo que hacen son los datos, que no hacen nada.

6.1.3 Acciones: funciones con efectos secundarios

Las funciones que tienen entradas o salidas implícitas, son *acciones*, también llamadas *funciones con efectos secundarios* o *funciones impuras*. Una acción es cualquier cosa que tenga un efecto en el mundo exterior a la función o que sea afectada por el mundo exterior.

Sus características principales son:

- Las acciones se implementan en el código utilizando funciones.
- Se propagan por el código: si una función llama a otra función que es una acción, ella misma se convierte en una acción.
- Las acciones dependen o pueden depender de cuándo se ejecutan (del orden de ejecución) o de cuantas veces se ejecutan (repetición).
- El significado de las acciones es el efecto que producen en el mundo exterior. Hay que asegurarse de que producen el efecto deseado.
- Las acciones son inevitables. De hecho, son la razón de ser de los programas.

La Programación Funcional propone diversas técnicas para gestionar de manera adecuada las acciones:

- Reducir el número de acciones a las estrictamente necesarias.
- Mantenerlas reducidas a su mínima expresión, extrayendo de ellas todos los elementos que sean cálculos o datos.
- Restringir las acciones a las interacciones con el exterior. En el interior, idealmente, todo son cálculos y datos.
- Reducir la dependencia del tiempo. Hacer que las acciones dependan menos de cuándo se jecutan o de cuántas veces se ejecutan.

6.2 Tipos de entradas y salidas de las funciones

Todas las funciones tienen entradas y/o salidas. Se pueden clasificar en implícitas y explícitas:

- *Entradas explícitas*: a través de los parámetros.
- *Salidas explícitas*: a través del valor devuelto.
- *Entradas o salidas implícitas*: cualquier otra forma de entrada o salida de información de la función.

Las entradas *explícitas* son los valores de los argumentos que recibe la función. Cualquier otra entrada de información a la función se considera una entrada *implícita*. Las salidas *explícitas* son los valores devueltos. Cualquier otra salida de información de la función se considera una salida *implícita* (ver Figura 14.1).

La existencia de entradas o salidas implícitas convierten una función en impura. Una *entrada implícita* puede ser leer el valor de una variable externa a la función, por ejemplo una variable global. También sería un entrada implícita la consulta de datos provenientes de una base de datos. Las *salidas implícitas* son cualquier información que salga de la función y que no sea el valor devuelto, por ejemplo, modificar el valor de una variable externa o enviar un email o escribir informacion en una base de datos o un fichero.

Cualquier entrada o salida implícita de información a la función convierte la función en impura, en una acción.

Si en una función se eliminan todas las entradas y salidas implícitas, se convierte en un cálculo. Las entradas implícitas, hay que convertirlas en parámetros y las salidas implícitas hay que convertirlas en valores devueltos.

6.3 Separar las cosas

Conviene recordar el concepto de *refactorizar*. Consiste en hacer modificaciones en el código organizándolo de otra manera sin alterar su comportamiento. Hay diversas técnicas de refactorización que se han estandarizado, por ejemplo, la denominada *extraer subrutina* consiste en extraer una parte de una función a otra función independiente.

En general, la refactorización da lugar a un código con más líneas, pero es más comprensible, más reusable y es más fácil hacer los test.

Una técnica de diseño reconocida consiste en *separar las cosas* (*pull things apart*). Ésta técnica también es conocida como el *Principio de Responsabilidad única*: que cada función haga solo una cosa y que la haga bien. Da lugar a un

código con más funciones y más pequeñas, pero facilita la reutilización y hace el código más fácil de comprender. En la Programación Funcional se promueve la utilización de funciones más pequeñas, separando las cosas. Los procesos complejos se consiguen mediante la *composición* de dichas funciones. La figura 14.2 trata de esquematizar está técnica.

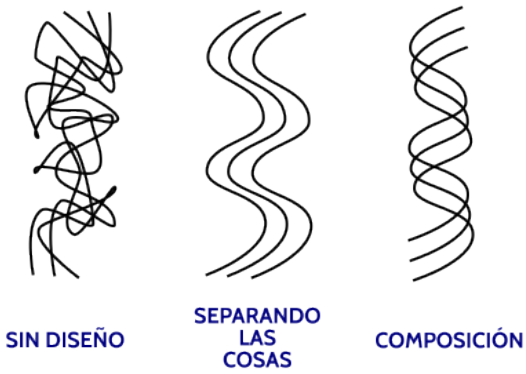

Figura 6.2: Visualización de la separación de diferentes cosas y su composición

6.4 Extraer los cálculos de las acciones

Observe la función *grabar_caudal()* que se muestra en el siguiente fragmento de código:

```
fn grabar_caudal() {
   let velocidad = leer_sensor();
   let caudal = 1.8 * velocidad.powf(3.2);
   save_caudal_to_database(caudal);
}
```

La función lee el valor de la velocidad de un sensor, realiza el cálculo del caudal asociado a esa velocidad y graba el resultado en una base de datos. La función es una acción: no tiene ningún argumento de entrada ni devuelve ningún resultado, pero tiene una entrada de datos implícita a través de la función *leer_sensor()* y una salida de datos implícita que graba datos en la base de datos utilizando la función *save_caudal_to_database()*.

Además, la función incluye un cálculo, que aquí se ha expresado mediante una operación relativamente sencilla, pero que podría ser más complicado.

Si se quisiera hacer algún test para comprobar que el cálculo se realiza de manera correcta, no sería fácil. Una opción sería habilitar el sensor correspondiente y la base de datos asociada mientras se hacen los test. Para comprobar que el resultado calculado es correcto, seguramente habría que consultar el valor grabado en la base de datos, pues la función no devuelve ningún valor. Si se quisiera probar el cálculo con diferentes valores de velocidad, habría que *trucar* el sensor, lo que puede resultar complicado.

El problema está en que se están mezclando dos cosas: una acción (en ralidad, dos) y un cálculo. La solución adecuada es extraer el cálculo a una función independiente, como se muestra en el siguiente código:

```
fn grabar_caudal() {
   let velocidad = leer_sensor();

   let caudal = calcula_caudal(velocidad);

   save_caudal_to_database(caudal);
}
fn calcula_caudal(velocidad: f64) -> f64 {
   let resultado = 1.8 * velocidad.powf(3.2);
   resultado
}
```

Ahora hay dos funciones: una acción y un cálculo. Es fácil realizar los test para comprobar que los cálculos se hacen de manera adecuada. Se puede probar la función *calcula_caudal()* para cualquier valor de entrada de manera sencilla.

Los pasos que se han seguido se podrían esquematizar de la siguiente forma:

- En la función original, seleccionar el fragmento de código a extraer.
- Ponerlo en una nueva función, sustituyéndolo en la función original por una llamada a la función recién creada. Es posible que haya que añadir algún parámetro.
- Identificar todas las entradas y salidas en la nueva función, convirtiendo las entradas en parámetros y las salidas en valores devueltos, que habrá que asignar a alguna variable en la función original.
- Los argumentos y lo valores devueltos de la función recién creadas tienen que ser inmutables, de forma que la nueva función sea un cálculo.

6.5 Ejemplo: eliminar entradas implícitas

En numerosas ocasiones, una misma función hace más de una cosa. Suponga que se tiene la siguiente función:

```
fn main() {
   facturar(100.0);
}
fn facturar(precio: f64) {
   let descuento = 0.10 * precio;
   println!("Precio      : {:.2}", precio);
   println!("Descuento   : {:.2}", descuento);
   println!("Precio neto : {:.2}", precio - descuento);
}
```

La función *facturar()* recibe como argumento el precio de un artículo, calcula el descuento a aplicar y muestra el resumen de la factura en pantalla. Está haciendo dos cosas: calcular el descuento y mostrar los resultados en pantalla. El hecho de mostrar los resultados en pantalla es un efecto secundario, una salida implícita que convierte a la función en una acción. Pero, además, tiene una entrada implícita de información. El valor 0.10 del porcentaje de descuento que se aplica es una especie de variable global, correspondiente a la lógica de negocio de la aplicación.

No es posible eliminar la acción, de hecho, es el objetivo del programa: mostrar la factura en pantalla, una vez que se ha aplicado el descuento. Es difícil diseñar un test para probar la función. Por una parte hay que probar que los cálculos son correctos y, por otra, que la salida de la función también es correcta. Tenga en cuenta que, aunque la salida se ha planteado simplemente mostrando unos mensajes en pantalla, podría tratarse de la actualización de ciertos registros en una base de datos o cualquier otra salida. En el caso de hacer la salida a una base de datos, para probar los cálculos, habría que habilitar y tener disponible la base de datos.

Lo correcto es separar el cálculo del descuento de la salida de resultados. Pero, además, la entrada implícita del porcentaje de descuento hay que convertirla en un parámetro de la función que calcula el descuento. Podría hacerse de la siguiente forma:

```
fn main() {
   let porcentaje_descuento = 0.10;
   let precio = 100.0;
   let descuento =
       calcula_descuento(precio, porcentaje_descuento);
   facturar(precio, descuento);
}
fn calcula_descuento(precio: f64, porcentaje: f64) -> f64 {
   precio*porcentaje
}
fn facturar(precio: f64, descuento: f64) {
   println!("Precio      : {:.2}", precio);
   println!("Descuento   : {:.2}", descuento);
   println!("Precio neto : {:.2}", precio - descuento);
}
```

Ahora, la función *calcula_descuento()* es una función pura, un cálculo. Sus únicas entradas se producen a través de los argumentos que recibe y su salida es el valor que devuelve. No importa cuándo se llame a la función o cuántas veces se la llame: si recibe los mismos argumentos siempre devolverá el mismo resultado. Es fácil plantear los test para probar que realiza los cálculos de manera adecuada.

Además, el porcentaje del descuento que se tiene que aplicar se ha independizado de cualquiera de las funciones. Está centralizado en una variable accesible para otras funciones del programa. Si la empresa decidiera cambiar su política de descuentos, solo tendría que modificar el valor de la variable en un sitio y se actualizaría la política en cualquier función que haga uso de ella.

Por otra parte, la función *facturar()* sigue siendo una acción, pero independiente de otras consideraciones. Si se decidiera cambiar la forma de salida de la pantalla a una base de datos, por ejemplo, solo habría que actuar en dicha función, despreocupándose del cálculo de los valores asociados. Aun siendo una acción, se han eliminado sus entradas implícitas, convirtiéndolas en parámetros.

En el ejemplo, se han realizado varias refactorizaciones:

- Extraer un cálculo de una acción.
- Convertir una entrada implícita en un parámetro.
- Separar las cosas en funciones independientes.

6.6 Inmutabilidad. Copy-on-write

Al inicializar una variable, se cambia su valor. Si en las funciones solo se inicializan variables locales, nada fuera de la función ve dichos cambios, salvo que lo vean a través del valor devuelto.

Ejemplo sin copy-on-write:

```
fn main() {
   let mut list = vec![1, 2, 3];
   list.push(4);
   assert_eq!(list, vec![1, 2, 3, 4]);
}
```

Sin copy-on-write, pero con función:

```
fn main() {
   let mut list = vec![1, 2, 3];
   modify_list(&mut list, 4);
   assert_eq!(list, vec![1, 2, 3, 4]);
}
fn modify_list(list: &mut Vec<i32>, x: i32) {
   list.push(x);
}
```

Con copy-on-write:

```
fn main() {
   let list = vec![1, 2, 3];
   let list = copy_on_write(list, 4);
   assert_eq!(list, vec![1, 2, 3, 4]);
}
fn copy_on_write(list: Vec<i32>, x: i32) -> Vec<i32> {
   let mut new_list = list;
   new_list.push(x);
   new_list
}
```

No se modifica la lista original, se transforma en una nueva lista que incorpora las modificaciones. No hay pérdida de rendimiento, la propiedad de los valores se va trasmitiendo, sin modificar su posición en memoria.

Variante usando un *RefCell*:

```
fn main() {
    let list = RefCell::new(vec![1, 2, 3]);
    let list = modify_cell(list, 4);
    assert_eq!(list.take(), vec![1, 2, 3, 4]);
}
fn modify_cell(list: RefCell<Vec<i32>>, x: i32)
-> RefCell<Vec<i32>> {
    let new_list = list;
    new_list.borrow_mut().push(x);
    new_list
}
```

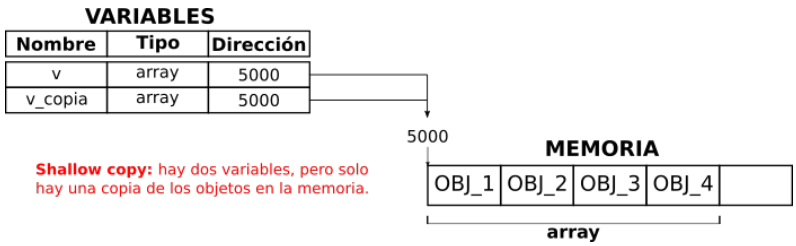

Figura 6.3: Esquema del mecanismo *shallow-copy* de un *array* en Java: se copian las referencias a los objetos, no los objetos en sí mismos. Al final hay dos variables apuntando a los mismos objetos en memoria.

CAPÍTULO 7

Colecciones

CAPÍTULO 8

Patrones de diseño en la Programación Funcional

CAPÍTULO 9

Grokking Simplicity

Contenido

9.1 Welcome to Grokking simplicity
9.2 Functional thinking in action
9.3 Distinguishing actions, calculations and data
9.4 Distinción entre acciones, cálculos y datos
9.5 Ejemplo
9.6 Extracting calculations from actions
9.7 Improving the design of actions
9.8 Staying inmutable in a mutable language
9.9 Staying inmutable with untrusted code
9.10 Stratified design (I)
9.11 Stratified design (II)
9.12 First-class functions (I)
9.13 First-class functions (II)
9.14 Functional iteration
9.15 Chaining functional tools
9.16 Functional tools for nested data
9.17 Isolating timelines
9.18 Sharing resources between timelines
9.19 Coordinating timelines
9.20 Reactive and onion architectures
9.21 The functional journey ahead

Apuntes del Libro «Grokking Simplicity», de Eric Norman [11]

9.1 Welcome to Grokking simplicity

9.2 Functional thinking in action

9.3 Distinguishing actions, calculations and data

Los programadores funcionales clasifican cualquier fragmento de código como acción, cálculo o datos. Esta clasificación puede recibir otras denominaciones, pero el concepto es el mismo.

- **Acciones:** son todo aquello que dependa de en qué momento se ejecuta o de cuántas veces se ejecuta. Por ejemplo, enviar un correo es una acción.
- **Cálculos:** son las tareas que solo dependen de los valores de entrada para generar un resultado. Los cálculos siempre devuelven el mismo resultado, si los valores de entrada son los mismos. Además, los cálculos nunca afectan a nada que esté fuera de ellos. Esto hace que los cálculos sean fáciles de testear y que su uso sea seguro, pues no hay que preocuparse de cuántas veces se utilicen o en qué orden sean invocados: si los parámetros de entrada son los mismos, el resultado será siempre el mismo.
- **Datos:** los datos son información registrada acerca de los acontecimientos. Tienen propiedades conocidas. Un mismo dato se puede interpretar de manera diferente según el contexto de ejecución en el que se encuadra. Por ejemplo, la factura de una cena la puede utilizar el cliente para llevar la cuenta de sus gastos mensuales o la puede utilizar el propietario del restaurante para determinar los gustos favoritos de sus clientes.

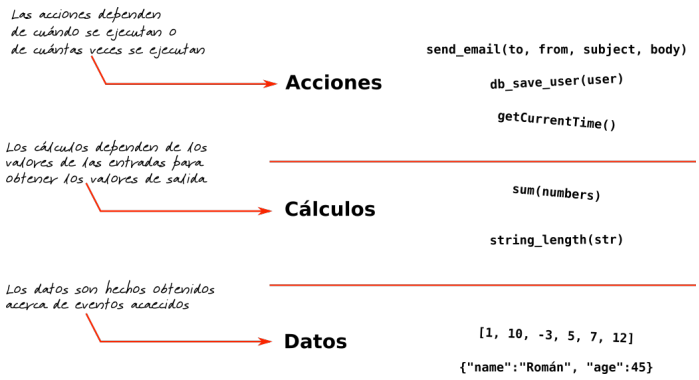

Figura 9.1: Ejemplos de código: Acciones, Cálculos y Datos

La proliferación de los sistemas distribuidos en los que intervienen diferentes dispositivos y peticiones cuasi simultáneas de información a las bases de datos,

el problema de organizar de manera adecuada el código se hace indispensable. En general, el código de las acciones será más difícil de comprender y de testear que el código de los cálculos, ya que estos últimos no dependen del número de veces que se invocan o del orden de dichas invocaciones. Es por ello que se hace importante separar en la medida de los posible el código que corresponde a las acciones del código que corresponde a los cálculos.

La programación funcional proporciona herramientas para el correcto tratamiento de cada una de estas categorías de código. En el caso de las acciones, es importante gestionar la forma en que cambia el estado de las variables del programa a lo largo del tiempo, garantizando el número de veces y el orden en el que se realiza cada acción. Los cálculos tienen su propia estrategia para comprobar su buen funcionamiento, a veces basada en técnicas matemáticas. En el caso de los datos, es importante organizarlos en estructuras que faciliten un acceso eficiente a la información que se quiere extraer de los mismos.

Hay dos técnicas fundamentales que permiten abordar los programas con un enfoque funcional:

- Distinguir en el código las acciones, de los cálculos y los datos.
- Utilizar abstracciones de primera clase.

A lo largo del curso se tratará de trasmitir el razonamiento funcional a la hora de abordar un problema de codificación. El objetivo es que las técnicas que se aprendan sean independientes del lenguaje que se utilice para programar y que sean de aplicación inmediata, tanto para la realización de un programa nuevo, como para refactorizar partes de un código ya existente.

Para clasificar el código en acciones, cálculos y datos es útil seguir los principios del *diseño estratificado*, separando el código en diferentes capas. Para organizar el orden en el que se ejecutan las acciones son de utilidad los *diagramas de tiempos* y la utilización de funciones de *primera clase*, que permiten utilizar otras funciones como parámetros o resultados.

En el diseño estratificado, se organiza el código en diferentes capas, ordenadas en función de la mayor o menor probabilidad de cambios en el código correspondiente a lo largo de la vida útil de la aplicación. Se suelen considerar tres capas principales:

- **Nivel técnico:** correspondería a la parte de la aplicación que tiene menos probabilidades de cambiar. Por ejemplo, el lenguaje de programación de la aplicación o las estructuras de datos que se utilizarán para almacenar la información.

- **Reglas del dominio de la aplicación:** si por ejemplo se está haciendo una aplicación para gestionar unos cultivos de hortalizas, las distancia optima a la que hay que poner las plantas en el terreno o la cantidad de humedad que necesitan corresponden al campo de conocimiento de dicho dominio técnico y es difícil que cambien durante la vida útil de la aplicación.
- **Reglas del negocio:** se consideran aquí reglas que vienen marcadas por la aplicación concreta que se esté desarrollando. En el ejemplo de los cultivos podrían ser los precios de los factores de producción o la disponibilidad de determinados recursos.

Cada capa de la aplicación se desarrolla sobre las demás y solo debe depender de las capas que hay situadas por debajo de ella. De esta forma, si se produce una modificación en algún elemento, se sabe que solo puede afectar a los elementos que estén situados en la misma capa o en las capas superiores. La Figura 17.2 muestra un ejemplo de organización en capas de una aplicación para gestionar cultivos.

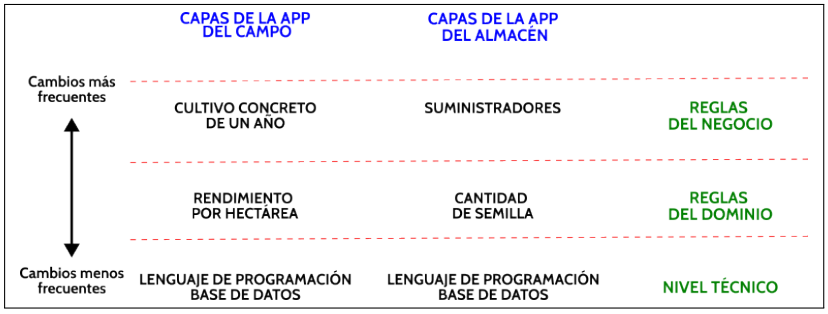

Figura 9.2: Esquema de capas en el diseño estratificado

9.3.1 Líneas de tiempos

En todas las aplicaciones hay que establecer el orden en el que se tienen que realizar las acciones. En aplicaciones sencillas, una distribución secuencial de las acciones puede ser suficiente. Pero, en sistemas distribuidos, en los que distintas tareas pueden correr a cargo de distintos componentes que pueden trabajar en paralelo o de forma concurrente, es importante establecer el orden en el que se tienen que realizar todas las acciones y los puntos en los que determinadas acciones no pueden ejecutarse si antes no se ha finalizado determinada acción anterior. Hay que cortar la línea de tiempos en algunos puntos para indicar que la ejecución no puede continuar hasta que se completen todas las tareas anteriores.

La Figura 17.3 muestra la línea de tiempos de una aplicación en la que, para poder ejecutarse las acciones *D* y *E* es necesario que primero se hayan completado las tareas llevadas a cabo por los componentes *A*, *B* y *C*.

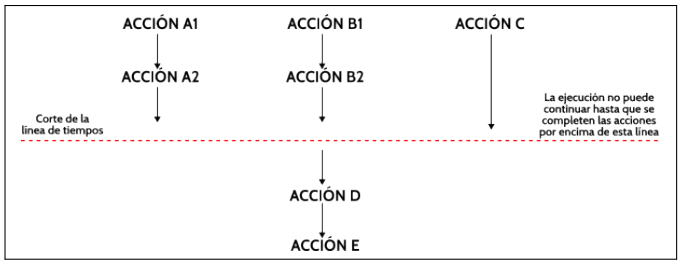

Figura 9.3: Cortando la línea de tiempo para garantizar el orden de ejecución de las acciones

La línea de tiempos ayuda a coordinar distintas acciones y a determinar los puntos en los que se pueden producir cuellos de botella durante la ejecución de los programas en sistemas distribuidos.

9.4 Distinción entre acciones, cálculos y datos

Antes de ponerse a codificar una aplicación nueva, conviene razonar .para distinguir qué elementos del programa serán acciones, cálculos o datos. En general, el orden de implementación consistirá en definir primero los datos, luego los cálculos y, por último, las acciones.

Cuando se está analizando un código ya existente, habrá que identificar igualmente las funciones que incluyen acciones. En muchos casos, una misma función incluirá acciones y cálculos. En esos casos, conviene refactorizar para separar las acciones de los cálculos.

Como se ha comentado, las acciones dependen de cuándo se ejecutan o de cuantas veces se ejecutan. Las funciones que implementan acciones se suelen denominar *funciones impuras* o *funciones con efectos secundarios*. Enviar un correo o leer datos de una base de datos serían ejemplos de acciones.

Por el contrario, los cálculos solo dependen del valor de los parámetros de entrada y no producen efectos secundarios. Se denominan *funciones puras* o *funciones matemáticas*. Calcular el máximo de una serie de números o comprobar si determinada dirección de correo es válida podrían ser ejemplos de funciones puras.

Se dice que los cálculos son *referencialmente trasparentes*. La trasparencia referencial significa que se puede sustituir el cálculo por el resultado sin que el

programa se vea afectado. Por ejemplo, si una función realiza la suma de dos números, en los lugares del programa donde se hace la llamada a la función se puede sustituir ésta por el resultado de la suma y el programa no se verá afectado.

Los datos son hechos.

9.4.1 Entradas y salidas implícitas

Los parámetros de una función son el procedimiento de *entrada explícita* de datos a la función. El valor devuelto por la función es la salida *explícita*. Cuando una función es impura, tiene entradas o salidas implícitas. Se denomina *entrada implícita* a la entrada de datos a la función que no procede de un parámetro. Se denomina *salida implícita* al valor que sale de la función sin hacerlo a través del valor devuelto por la misma.

El código del ejemplo muestra una función denominada *contador_impuro()* que no tiene parámetros ni devuelve ningún valor. En cambio, la función recibe como entrada implícita el valor de un contador existente en la base de datos, a través de un método llamado *get_contador_from_database()*, y realiza una salida también implícita reescribiendo en la base de datos el valor incrementado de dicho contador a través del método *update_contador_in_database()*. La salida implícita de la función es, además, un efecto secundario de la misma, convirtiendo a dicha función en una *acción*.

Ejemplo 9.1 Entradas y salidas implícitas

```
fn contador_impuro() {
   let contador = get_contador_from_database(); // Entrada implícita
   update_contador_in_database(contador+1); // Salida implícita
}
```

El acceso a variables globales dentro de una función es una forma de entrada o salida implícita.

En la medida de lo posible, hay que evitar las entradas y salidas implícitas, pues complican la trazabilidad y la facilidad de testeo de las funciones. En muchas ocasiones, las entradas implícitas se pueden sustituir por parámetros de la función. De la misma forma, las salidas implícitas es posible sustituirlas por valores devueltos por las funciones.

9.5 Ejemplo

Ejemplo 9.2 Ejemplo carrito

```
struct Producto {
   name: String,
   precio: f64,
}
fn main() {
   let mut carrito = Vec::<Producto>::new();
   let mut total_compra: f64 = 0.0;
   let producto = Producto{name: "Sandalias".to_string(), precio: 12.5
   add_producto(producto, &mut carrito, &mut total_compra);
}
fn add_producto(producto: Producto, carrito: &mut Vec<Producto>, total
   carrito.push(producto);
   calc_total_carrito(carrito, total_compra);
}
fn calc_total_carrito(carrito: &mut Carrito, total_compra: &mut f64) {
   *total_compra = 0.0;
   for producto in &carrito.items {
      *total_compra = *total_compra + producto.precio;
   }
   actualiza_web_total_compra( *total_compra);
}
fn actualiza_web_total_compra(total_compra: f64) {  }
```

9.6 Extracting calculations from actions
9.7 Improving the design of actions
9.8 Staying inmutable in a mutable language
9.9 Staying inmutable with untrusted code
9.10 Stratified design (I)
9.11 Stratified design (II)
9.12 First-class functions (I)
9.13 First-class functions (II)
9.14 Functional iteration
9.15 Chaining functional tools
9.16 Functional tools for nested data
9.17 Isolating timelines
9.18 Sharing resources between timelines
9.19 Coordinating timelines
9.20 Reactive and onion architectures
9.21 The functional journey ahead

CAPÍTULO 10

Functional Programming made easier (Scalfani)

10.1 1.- Discipline is freedom

10.1.1 Global State

El uso de varibles globales conlleva determinados inconvenientes:

- Cualquiera desde cualquier módulo puede cambiar el valor de las variables globales.
- Se producen acoplamientos de las variables globales entre sí y de unos módulos con otros.
- En programación concurrente no hay garantías respecto de la modificación de las variables.
- Colisión de nombres entre las variables globales y otros identificadores en cualquier módulo.

En el caso de la programación orientada a objetos se produce la misma circunstancia con el patrón *Singleton*.

Los lenguajes de PF prohíben la existencia de variables globales. Rust permite solo la existencia de constantes globales.

10.1.2 Mutable State

La posibilidad de que las variables puedan cambiar de valor también tiene algunos inconvenientes. Por ejemplo, es más difícil razonar sobre el código, pues los valores que pueden cambiar pueden hacer cambiar también la semántica del programa, o hacer el código más frágil.

En los lenguajes funcionales, las variables son inmutables y ello da lugar a algunas consecuencias.

- Las expresiones del tipo $x = x + 1$ no tienen sentido. Una vez que se asigna un valor a x, no se puede cambiar.

- Las asignaciones como $x = 20$ son *expresiones referencialmente trasparentes*, esto es, en cualquier parte del programa se puede utilizar indistintamente x o 20 con la seguridad de que el programa seguirá funcionando igual. La trasparencia referencial permite una evaluación *lazy* de la sustitución de x por su valor en el código.

Si las variables no pueden cambiar de estado, no es posible hacer bucles. En los lenguajes funcionales, los bucles se sustituyen por la recursividad. Cualquier bucle se puede ejecutar mediante recursividad y viceversa.

Por ejemplo, la definición matemática del factorial de un número se podría hacer de la siguiente forma:

$$n! = 1.2...(n-2).(n-1).n \quad \forall n \in \mathbb{N} \tag{10.1}$$

Con esta definición, parecería inmediato resolver el problema con un bucle, como se hace en el Ejemplo 18.1.

Ejemplo 10.1 Factorial calculado con un bucle

```
fn factorial_bucle(n: u32) -> u32 {
   let mut prod = 1;
   for i in 1..=n {
      prod = prod*i;
   }
   prod
}
```

Si en la Expresión 18.1 se cambia el orden del producto, los términos se podrían agrupar de la siguiente manera:

$$n! = n.(n-1)...2.1 \quad = n.(n-1)! \quad \forall n \in \mathbb{N} \tag{10.2}$$

Con esta definición surge el problema de calcular $0!$, pero su valor se puede deducir. De la Expresión 18.1 se sabe que $!1 = 1$. Se podría operar de la siguiente forma:

$$\begin{aligned} 1! &= 1 \\ 1! &= 1.(1-1)! = 1.0! \\ 1 &= 1.0! = 0! \end{aligned}$$

Con lo que finalmente queda que:

$$0! = 1 \tag{10.3}$$

Una vez calculado el valor de $0!$, se puede proceder a definir el factorial de cualquier número natural con la siguiente definición recursiva:

$$\begin{aligned} 0! &= 1 \\ n! &= n.(n-1)! \quad \forall n \in \mathbb{N} \end{aligned} \tag{10.4}$$

Esta definición de factorial se podría codificar como se hace en el Ejemplo 18.2.

Ejemplo 10.2 Factorial calculado de manera recursiva

```
fn factorial_recursivo(n: u32) -> u32 {
   match n {
      0 => 1,
      _ => n*factorial_recursivo(n-1)
   }
}
```

10.2 Variables globales

Vamos a ver en primer lugar cómo no se pueden usar las variables globales en Rust. Podríamos estar tentados de usar sentencias *let*, como en la variables locales de cualquier función:

```
use chrono::Utc;

let START_TIME: String = Utc::now().to_string();

fn main() {
   ...
}
```

El código anterior no compila, no se pueden usar asignaciones *let* en el ámbito global, pues la instrucción *let* crea una variable en la memoria stack, que no se inicializa hasta que se ejecuta el programa. Las variables globales solo se

pueden crear utilizando las cláusulas *const* o *static*, que utilizan la memoria del segmento de datos (*data segment*) del programa.

Tampoco se puede compilar la siguiente inicialización:

```
use chrono::Utc;

static START_TIME: String = Utc::now().to_string();

fn main() {
   ...
}
```

El compilador avisa que no se pueden utilizar funciones no constantes en la inicialización de variables globales. No se puede ejecutar ningún tipo de código antes de que el programa comience. El valor de una variable global debe ser conocido en el momento de la compilación, antes de la ejecución.

Tampoco valdría declarar la variable global y tratar de asignarle valor en *main()*:

```
use chrono::Utc;

static START_TIME: String;

pub fn main() {
   START_TIME =  = Utc::now().to_string();
   println!("{}", START_TIME);
}
```

El código anterior tampoco compila, el compilador nos avisa de que hay que asignar algún valor a la variable *static*. Podríamos pensar en asignarle un valor *None*

```
use chrono::Utc;

static mut START_TIME: Option<String> = None;

pub fn main() {
   START_TIME = Some(Utc::now().to_string());
   println!("{}", START_TIME);
}
```

Pero el código anterior tampoco compila, el compilador nos avisa de que una variable global mutable solo se puede utilizar dentro de un bloque inseguro *unsafe*. Finalmente, si encerramos el código dentro de *main()* en un bloque *unsafe*, sí que compila:

```
use chrono::Utc;

static mut START_TIME: Option<String> = None;

pub fn main() {
   unsafe{
      START_TIME =  Some(Utc::now().to_string());
      println!("{}",  START_TIME.clone().unwrap());
   };
}
```

Ahora, el código compila y se puede ejecutar, pero no parece una forma muy cómoda de utilizar, aunque en algunas ocasiones pudiera ser útil.

Vamos a ver otro problema relacionado con la extensión del dominio de las variables. En un programa como el anterior, no se necesitaría declarar la variable como global, se podría declarar dentro de la función *main()*. Pero suponga que lo que se quiere es utilizar la variable en un hilo diferente creado dentro de *main()*

```
use chrono::Utc;

pub fn main() {
   let start_time = Utc::now().to_string();

   let thread_1 = std::thread::spawn(||{
      println!("Started {}, called thread 1 {}", &start_time, Utc::now());
   });

   thread_1.join().unwrap();
}
```

Si tratamos de compilar este código, el compilador nos dirá que el hilo *thread_1* podría vivir más tiempo que la variable *start_time* que se ha creado en el marco de datos en el stack de la función *main()* y no está garantizado que la variable sobreviva lo suficiente. Nosotros, viendo el código, sabemos que al hacer el *join()* antes de salir de *main()* estamos garantizando esa supervivencia, pero el compilador no nos deja compartir con otros hilos variables que no tengan una vida útil *&static*.

Hay un par de soluciones posibles sin utilizar variables globales. La primera sería clonar la variable *start_time* y pasar a la closure la propiedad del valor clonado:

```
pub fn main() {
   let start_time = Utc::now().to_string();
   let cloned_start_time = start_time.clone();
   let thread_1 = std::thread::spawn( move ||{
      println!("Started {}, called thread 1 {}", &cloned_start_time, Utc::
   });
   thread_1.join().unwrap();
}
```

Esta solución puede servir para una variable de cadena de caracteres como la anterior, pero si hubiera que clonar una variable de mayor tamaño podría no ser la solución óptima. En esos casos se podría envolver la variable con un puntero *Arc*:

```
use chrono::Utc;
use std::sync::Arc;

pub fn main() {
   let start_time = Arc::new(Utc::now().to_string());
   let cloned_start_time = Arc::clone(&start_time);
   let thread_1 = std::thread::spawn(move ||{
      println!("Started {}, called thread 1 {}", &cloned_start_time,
   });

   thread_1.join().unwrap();
}
```

Si además se necesitara mutabilidad interior de la variable, se podría envolver en un `Arc<Mutex<String>>`.

10.2.1 Valor conocido en tiempo de compilación

Cuando el valor de la variable global se conoce en tiempo de compilación, hay básicamente dos soluciones:

- **const:** valores constantes que se conocen en tiempo de compilación. No permiten la mutabilidad interior. El compilador resuelve sustituyendo el valor en línea (inline)[1].
- **static:** las variables reciben un espacio de memoria en el segmento de datos. Es posible la mutabilidad interior.

Si se necesita mutabilidad interior, para los tipos primitivos se pueden utilizar valores *atomic* [2] y, para tipos más complejos, se pueden usar *locks* en la forma *read-write lock*, *RwLock* o en la forma *mutual exclusion lock*, *Mutex*.

Si lo que se necesita es calcular el valor en tiempo de ejecución, las soluciones con *const* y *static* no sirven.

[1] Conviene consultar el apartado de la cláusula *const* en la documentación en `https://doc.rust-lang.org/std/keyword.const.html` y del concepto de *expresiones constantes* en `https://doc.rust-lang.org/reference/const_eval.html`

[2] Ver libro en líne de Mara Bos `https://marabos.nl/atomics/`

CAPÍTULO 11

Railway Oriented Programming

El *Railway Oriented Programming* es un término acuñado por Scott Wlaschin que propone un modelo del tratamiento de errores en la programación funcional.

- Railway Oriented Programming: `https://fsharpforfunandprofit.com/rop/`
- Monoids without tears: `https://fsharpforfunandprofit.com/posts/monoids-wi`

11.1 Monoides

Morfismo: en varios campos de las matemáticas, se llaman *morfismos* (u *homomorfismos*) a las aplicaciones entre estructuras matemáticas que preservan la estructura interna. Por ejemplo, en teoría de conjuntos, los morfismos son las aplicaciones entre conjuntos; en álgebra lineal, las transformaciones lineales; y en topología, las funciones continuas. En *teoría de las categorías*, el morfismo tiene una noción más general.

Categoría: una categoría viene dada por dos tipos de datos: una clase de objetos y, para cada par de objetos X e Y, un conjunto de morfismos desde X hasta Y. En el caso de una categoría concreta, X e Y son conjuntos de cierto tipo y un morfismo *f* es una función desde X a Y que satisface alguna condición.

Los morfismos se representan frecuentemente como flechas entre los objetos. Esto origina la notación:

$$f : X \to Y \tag{11.1}$$

CAPÍTULO 12

NaN an Inf

https://stackoverflow.com/questions/14682005/why-does-division-by-zero

12.1 Artículos para recomendar a los alumnos

The Mediocre Programmer's Guide to Rust:

https://www.hezmatt.org/~mpalmer/blog/2024/05/01/the-mediocre-programmers-guide-to-rus
html

CAPÍTULO 13

Funciones de orden superior, closures e iteradores

En matemáticas y en ciencias de la computación, se denominan funciones de orden superior a las que cumplen al menos una de las siguientes condiciones:

- *Tomar una o más funciones como parámetro de entrada.*
- *Devolver una función como salida.*

En matemáticas, un ejemplo lo podría constituir la función derivada, que toma una función como parámetro y devuelve otra función.

En programación funcional, las funciones son consideradas elementos de primera clase, esto es, pueden usarse como argumentos de otras funciones, como valores devueltos y también asignarse a un símbolo o variable. La mayoría de los lenguajes han adoptado ya esta premisa y dan un tratamiento de primera clase a las funciones.

En este capítulo se va a hablar de funciones, de cómo asignarlas a variables y de cómo utilizarlas como argumento o como valor devuelto de otras funciones.

También se hablará de las closures, que son un tipo especial de funciones definidas dentro de otras funciones, que pueden acceder al contexto en el que se han definido.

Por último, se hablará de los iteradores y cómo encadenar procesos utilizando iteradores y funciones de orden superior.

13.1 Funciones de primera clase

En programación, se denominan *elementos de primera clase* a los elementos del lenguaje que se pueden asignar a variables y se pueden utilizar como argumentos o como valor devuelto por otras funciones.

No todos los elementos del lenguaje son elementos de primera clase. Por ejemplo, en muchos lenguajes, los operadores aritméticos como el $+$, el $-$ y otros, no son de primera clase. También sucede algo parecido con algunas construcciones, como los bucles o las bifurcaciones.

Es importante entender la diferencia entre el concepto de *declaración* y el de *expresión*. Un elemento de un lenguaje se considera que es una *declaración*

cuando no devuelve ningún valor; una *expresión*, en cambio, es un elemento que devuelve un valor.

En los lenguajes funcionales se da prioridad a las expresiones que devuelven valores. Así se hace en Rust. Algunas construcciones que en otros lenguajes son declaraciones, en Rust son expresiones. Es el caso de las bifurcaciones *if* o de los bucles *loop, for* o *while*, como se indicó en los apartados 3.12 y 3.13. En cambio, el operador *let* es una declaración que no devuelve ningún valor.

Se podría realizar un código como el siguiente:

```
fn main() {
   let x = 2;
   println!("{}", suma(if x==2 {3} else {4}, 5));

   let mut n = 1;
   println!("{}", loop {
      n=n+1;
      if n>3 {
         break 10
      }
   });
}
fn suma(x: i32, y: i32) -> i32 {
   x+y
}
```

En el programa anterior, se utiliza una bifurcación *if* como argumento de la función *suma()* y un bucle *loop* como argumento de la macro *println!()*. Observe que, en realidad, el argumento es el valor que devuelven dichas expresiones.

Actualmente, numerosos lenguajes se han adherido a la línea marcada por los lenguajes funcionales y consideran las funciones como elementos de primera clase, esto es, permiten que las funciones se puedan asignar a variables y se puedan utilizar como parámetros o como valor devuelto por otras funciones. En Rust, las funciones son objetos de primera clase.

Se denominan *funciones de primer orden* o también *funciones ordinarias* aquellas en las que ni los parámetros ni el valor devuelto son otras funciones. Por contra, las *funciones de orden superior* son aquellas en las que o bien alguno de los parámetros o bien el valor devuelto son una función.

Hay varias maneras de definir funciones en Rust:

- Dentro de un módulo o fichero *.rs*, con la visibilidad que le corresponda y acceso a otras funciones y a sus variables locales .
- Dentro de otra función o bloque de código, con visibilidad restringida a dicho bloque y acceso a otras funciones, pero no a variables del contexto del bloque en el que se ha definido.
- Cómo función asociada a un tipo de datos personalizado, como es el caso de las estructuras y las enumeraciones.
- Como closure definida dentro de otra función o bloque de código. Su visibilidad estará restringida al bloque en el que se ha definido, pero las closures sí que tienen acceso a las variables existentes en el contexto del bloque en el que se haya definido.

Ya se comentó este tema en el Apartado 3.14. También se habló allí de los tres tipos de funciones que considera Rust, en función del uso que hacen de la propiedad de los parámetros: el tipo *Fn*, para funciones ordinarias, el tipo *FnMut*, para funciones con parámetros mutables y el tipo *FnOnce* para funciones que solo se pueden invocar una vez.

13.1.1 Asignación de funciones a variables

Cualquiera de los tipos de funciones presentes en Rust se pueden asignar a una variable. Luego, se puede utilizar la variable como si fuera la función.

El siguiente código asigna a una variable de nombre *raiz2* la función *sqrt* de los números *f64*. Como resultado, la variable *raiz2* es del tipo *fn(f64) ->f64*, esto es, una función que recibe un argumento del tipo *f64* y devuelve un valor *f64*. Observe la forma de calcular la raíz cuadrada utilizando la variable así definida:

```
fn main() {
   let raiz2  = f64::sqrt;
   let y = raiz2(4.0); // Equivale a 4.0f64.sqrt()
   println!("{}", y);
}
```

Se podría asignar una función previamente codificada, como se hace en el siguiente código:

```
fn main() {
   let f = saludo;
   f(); // Imprime Hola, ¿qué tal?
}
fn saludo() {
   println!("Hola, ¿qué tal?");
}
```

En este caso, la variable *f* es del tipo *fn()*, esto es, del tipo de las funciones que no reciben argumentos y no devuelven ningún valor.

Las closures también se pueden asignar a variables, como se hace en el siguiente ejemplo:

```
fn main() {
   let cuadrado = |x| x*x;
   println!("{}", cuadrado(3.0)); // Imprime 9
}
```

Este tipo de funciones que no se definen dentro de un bloque *fn* con nombre, se denominan *funciones anónimas*. En este caso, la función anónima se ha asignado a la variable *cuadrado*.

También se suele decir que la closure se ha codificado *inline*. El término *inline* se refiere a que la codificación de la función se ha hecho dentro de la línea de código de otra instrucción. Es habitual también codificar closures *inline* cuando se utilizan como parámetros de otras funciones.

13.1.2 Utilización de funciones como parámetros de otras funciones

Observe el código siguiente:

```
fn main() {
   let pi = std::f64::consts::PI;
   println!("{}", operador(f64::sin, pi/2.0)); // Imprime 1
   println!("{}", operador(f64::cos, pi)); // Imprime -1
   println!("{}", operador(|x| x*x, 4.0)); // Imprime 16
   println!("{}", operador(duplica, 5.0)); // Imprime 10
}
fn operador(f:fn(f64) -> f64, x: f64) -> f64 {
   f(x)
}
fn duplica(x: f64) -> f64 {
   x*2.0
}
```

En el código anterior, la función *operador* es una función de orden superior que se ha definido con la siguiente signatura:

```
fn operador(f:fn(f64) ->f64, x: f64) ->f64
```

Cualquier función con un único parámetro del tipo *f64* y que devuelve un valor del tipo *f64*, se podrá utilizar como argumento en la llamada a la función *operador()*. La función *operador()* devolverá el resultado de aplicar la función *f* que recibe como primer argumento al valor *x* del tipo *f64* que recibe como segundo argumento.

En algunos lenguajes, a la función que se pasa como argumento, para que la función de orden superior la invoque, se le denomina *función de callback*.

13.1.3 Uso de funciones como valor devuelto por otras funciones

En el siguiente código se define una función de orden superior, llamada *operador*, que recibe un número entero como argumento. Si el valor recibido es mayor que cero, *operador()* devuelve la función *f64::sin*; en caso contrario, devuelve la función *f64::cos*.

```
fn main() {
   let pi = std::f64::consts::PI;
   let seno = operador(3);
   println!("{}", seno(pi/2.0)); // Imprime 1
   let coseno = operador(-2);
   println!("{}", coseno(pi)); // Imprime -1
}
fn operador(n: i32) -> fn(f64) -> f64 {
   if n>0 {f64::sin} else {f64::cos}
}
```

Otra forma de indicar el tipo de función que se pasa como parámetro o que se devuelve como resultado es utilizar la cláusula *impl* y el nombre del *Trait*, que en este caso es parecido pero poniendo *Fn* en mayúsculas:

```
fn operador(n: i32) -> impl Fn(f64) ->f64
```

13.2 Errores de ejecución

El mecanismo de excepciones es la forma habitual que ofrecen muchos lenguajes para gestionar los errores que se producen durante la ejecución de los programas. Pueden ser errores debidos a la entrada de un dato incorrecto, al intento de acceso a un fichero que no existe, el intento de acceder a una página web sin tener conexión u otros tipos de errores de ejecución. En muchas ocasiones, el error de ejecución proviene de la existencia de un puntero nulo en alguna parte del programa.

El invento del puntero nulo se atribuye a Tony Hoare, durante la implementación de los tipos de datos para el lenguaje ALGOL W, a mediados de la decada de 1960. Años después, él mismo lo llamó *el error de los mil millones de dólares*[1]:

> *"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."*

[1] Esta cita se ha tomado de la Wikipedia

En Rust no existe el puntero *null*. Tampoco existe un mecanismo de excepciones para el tratamiento de los errores de ejecución. En Rust se distingue entre dos tipos de errores de ejecución:

- *Errores recuperables*: por ejemplo, se pide al usuario el nombre del fichero que se pretende abrir y se comprueba que dicho fichero no existe. En esos casos, lo que se puede hacer es informar del error al usuario y volver a pedirle el nombre del fichero.
- *Errores no recuperables:* por ejemplo, se necesita acceder a un dispositivo y el dispositivo no existe. En estos casos se interrumpe la ejecución del programa, quizás imprimiendo un mensaje que informa al usuario del error que se ha producido.

En Rust, cuando el error da lugar a la interrupción del programa se suele decir que el programa *ha entrado en pánico*, por analogía con el nombre de la macro *panic!()* que se suele utilizar para forzar interrupciones.

La macro *panic!()* recibe un mensaje de texto como argumento, interrumpe el programa y muestra en la consola dicho mensaje junto con otras informaciones.

Puede probar el siguiente código:

```
fn main() {
    panic!("¡Programa interrumpido!")
}
```

Para gestionar los errores recuperables se suelen utilizar las enumeraciones *Option<T>* y *Result<T>*, en la forma que se va a explicar en los siguientes apartados.

13.3 El *enum Option*

La enumeración *Option* tiene la siguiente declaración:

```
pub enum Option<T> {
    None,
    Some(T),
}
```

El tipo *T* puede ser cualquiera. Es fácil definir un valor del tipo *Option*. Por ejemplo, el siguiente código declara dos variables del tipo *Option<u32>*, una con cierto valor y otra sin valor:

```
let n = Some(3u32);
let m: Option<u32> = None;
```

Observe que, en el caso de la variable *n* que sí tiene valor, el tipo de datos viene definido por el valor en sí, mientras que en el caso de la variable *m* hay que especificr el tipo de datos del *Option*.

El tipo *Option<T>* se utiliza para encapsular un valor que puede no existir. Procede de los lenguajes funcionales y del concepto de *mónada*. En estos lenguajes es habitual encontrar el tipo *Maybe* (“tal vez un”), representando un valor que es o un valor del tipo *T*, o ninguno.

Los métodos *is_some()* e *is_none()* permiten saber si determinada variables del tipo *Option* es la variante *Some* o la variante *None*:

```
fn main() {
   let n = Some(3);
   let m: Option<u32> = None;
   assert!(n.is_some());
   assert!(m.is_none());
}
```

Cuando la variante de la variable es *Some*, se puede extraer su valor con el método *unwrap()*:

```
let n = Some(3);
assert_eq!(n.unwrap(), 3);
```

La función *unwrap()* interrumpe la ejecución del programa con *panic!*, si la variante es *None*. Por ello, se suele utilizar más el método *unwrap_or()*, que proporciona un valor del mismo tipo que el contenido en el *Option* como valor devuelto en el caso de que la variante sea *None*:

```
assert_eq!(Some("car").unwrap_or("bike"), "car");
assert_eq!(None.unwrap_or("bike"), "bike");
```

También existen unos métodos *unwrap_or_else()* y *unwrap_or_default()*, que puede consultar en la documentación del tipo *Option*.

Es habitual utilizar el tipo *Option* como resultados de funciones. Suponga una función que reciba dos parámetros enteros y devuelva la división de uno entre el otro. En Java, la solución podría ser la siguiente:

```
static int dividir(int num, int den) {
   return num/den;
}
```

En Java, si el denominador es cero, se lanzará una excepción y se interrumpirá el programa. En principio hay dos posibilidades para gestionar una función de ese tipo: capturar la excepción o interceptar los denominadores cero. [2].

Como se ha dicho, Rust no dispone de tipo *null* ni de ningún mecanismo de excepciones para la gestión de errores. En Rust, en cambio, se puede imponer que la función *dividir()* devuelva un *Option<i32>* en lugar de un valor entero. El código podría ser el siguiente:

```
fn main() {
   let n = dividir(3, 0);
   assert_eq!(n, None);
}
fn dividir(num: i32, den: i32) -> Option<i32> {
   match den {
      0 => None,
      _ => Some(num/den)
   }
}
```

Una ventaja de que la función devuelva un *Option* es que el programa está obligado a gestionar la posibilidad de que el valor devuelto no exista. En el caso

[2]No se podría hacer que la función devuelva *Infinity* pues no existe para valores del tipo *int*.

del programa Java, nada indica en la signatura de la función ni en el cuerpo de la misma que cabe la posibilidad de que se genere una excepción. Ello podría dar lugar a errores de ejecución no previstos. En el caso resuelto con Rust, el programa que llama a la función sabe que puede obtener como resultado la variante *None* y el compilador le obliga a gestionar los dos posibles resultados[3].

La ventaja de devolver resultados de funciones con *Option* es más evidente cuando hay que encadenar varias funciones en las que una o más de ellas pueden devolver resultados no válidos. En el Apartado 13.6 se mostrarán ejemplos de encadenamiento de funciones con iteradores.

13.4 Gestión de errores con *Result*

Como se ha mencionado, en *Rust* no existe el mecanismo de excepciones que se utiliza en otros lenguajes para resolver las situaciones en las que una operación pueda dar lugar a un error de ejecución. Tampoco existe un valor *null*. El módulo *result* de la librería *std* de *Rust* proporciona la enumeración *std::result::Result* para gestionar dichas situaciones. Cuando una operación pueda dar lugar a un resultado correcto o erróneo, lo que se hace es devolver una instancia de *Result*.

La enumeración *std::result::Result* se define de la siguiente manera:

```
pub enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

Las dos variantes de esta enumeración son:

- *Ok(T)*: representa el éxito de una operación y proporciona un valor del tipo de datos genérico *T*.
- *Err(E)*: representa que se ha producido algún error en la operación y proporciona un valor del tipo de datos genérico *E*.

Se va a desarrollar a continuación una función para dividir dos números del tipo *f64*. Para resolver la situación de que el divisor sea cero, la función devuelve un valor de la enumeración *Result* que se acaba de describir:

[3] Java ofrece también un tipo *Optional<T>* que, junto a otros tipos, permite acometer éste y otros aspectos de la programación funcional.

```
fn div(x: i32, y: i32) -> Result<i32, ()> {
   match y {
      0 => Err(()),
      _ => Ok(x / y)
   }
}
fn main() {
   let d = div(10, 0);
   match d {
      Ok(value) => println!("{}", value),
      Err(()) => println!("Error, denominador cero")
   }
}
```

Como se verá en el apartado 13.5, otra forma de gestionar los errores devueltos en un *Result* es utilizar el operador *?* para propagarlos hacia arriba, hacia la función que llama.

Un problema asociado a los lenguajes que utilizan el mecanismo de excepciones es que se pueden capturar las excepciones e ignorarlas. El tipo *Result* está definido con la anotación *#[must_use]*, que indica al compilador que se emita un mensaje de aviso si se ignora el resultado de una función que devuelva un *Result*. El lector puede probar a sustituir el código de la función *main()* del ejemplo anterior por el siguiente:

```
fn main() {
   div(10.0, 0.0);
}
```

Si ahora trata de compilar el programa, el compilador emitirá un aviso de que el *Result* que devuelve la función no se ha utilizado.

El lector puede comprobar también que, si el resultado de la función *div()* se asigna a una variable, no se genera el aviso *must_use*. Pruebe a compilar el siguiente código:

```
fn main() {
   let _x = div(10.0, 0.0);
}
```

En el código anterior, el nombre de la variable se ha iniciado con un guion bajo. Si no se hace así, el compilador genera otro aviso, el de que la variable *x* no se utiliza. Es una demostración más de las numerosas comprobaciones que hace el compilador de *Rust*.

13.5 El operador *?*

Una manera habitual de tratar el objeto *Result* que devuelven algunos métodos es mediante una construcción *match*, como se ha visto en el apartado anterior. Dentro de una función que devuelva un *Result* o un *Option*, se puede sustituir dicha construcción por el operador *?*, escrito a continuación de la función que devuelve el *Result*. Con el operador *?*, si el resultado es *Ok*, el operador *?* hace el *unwrap()*; si el resultado es *Err*, se vuelve de la función en la que esté, devolviendo el error. De esta manera, se deja la gestión del error para la función que llamó. Si se está dentro de *main()*, se devuelve al sistema operativo. Este mecanismo se denomina *propagación de los errores hacia arriba*. El siguiente ejemplo utiliza el operador *?*:

```
fn main() -> Result<(), ()> {
   let _d = div(10.0, 0.0)?;
   Ok(())
}
```

Como se ve en el código anterior, ahora la función *main()* tiene que devolver un *Result*. Si se ejecuta el programa anterior, la división genera un error, que se muestra en el sistema operativo como *Error: ()*. En el ejemplo, si no hay error, se devuelve la variante *Ok()* de *Result*.

13.6 Iteradores

Un *Iterador* es un objeto que permite recorrer ordenadamente los elementos de una fuente de datos. En muchas ocasiones, la fuente de datos es una colección de objetos, pero también se pueden utilizar iteradores para leer ficheros línea a línea o para leer streams procedentes de un puerto de comunicaciones, por ejemplo.

Cuando un tipo de datos implementa el *trait Iterator*, se convierte en una especie de colección, que es posible recorrer de manera ordenada. Por ello, es habitual referirse a los elementos de dicho tipo de datos como *los elementos del iterador*, como si se tratase de una colección.

El uso de iteradores es un patrón de diseño muy utilizado en la programación funcional y que se ha incorporado progresivamente en todos los lenguajes de programación. Su origen se marca en el lenguaje CLU del año 1973 [10].

Técnicamente, un iterador en Rust es un tipo de datos que implementa el *trait Iterator*, que impone la existencia del método *next()*. Este método, si hay más elementos por iterar, devuelve *Some(E)*, donde *E* es el tipo de datos del iterador; si no quedan más elementos por iterar, devuelve *None*.

Las colecciones que ofrece el lenguaje Rust —arrays, vectores, hashmaps y otras— permiten obtener un iterador sobre los elementos de la colección mediante el método *into_iter()*, o bien un iterador sobre referencias a los elementos de la colección utilizando el método *iter()*. La diferencia fundamental entre ambos métodos es que, si se utiliza el método *into_iter()*, se *consume* la colección, pasando la propiedad al proceso que la esté utilizando; el método *iter()*, en cambio, utiliza referencias y no *consume* la propiedad de la colección.

El siguiente ejemplo utiliza el método *iter()* con un vector. Tras recorrer el vector, los elementos de la colección siguen estando disponibles:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];
   for num in v.iter() {
      println!("{}", num);
   }
   println!("{}", v[0]);
}
```

En cambio, si en el ejemplo anterior se utiliza el método *into_iter()*, se consume el vector en el bucle y los elementos de la colección ya no estarán accesibles después de la ejecución del mismo:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];
   for num in v.into_iter() { // into_iter() mueve la propiedad
      println!("{}", num);
   }
   println!("{}", v[0]); // Error, v se movió al bucle
}
```

Es posible utilizar un bucle *for* para recorrer una colección sin llamar explícitamente a ninguno de los dos métodos. El problema será similar: si se recorre la colección, se pasa la propiedad al bucle, mientras que si se recorre utilizando una referencia, la colección no se consume en el bucle.

El siguiente ejemplo utiliza una referencia a la colección y los elementos del vector siguen disponibles tras finalizar el bucle:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];

   for num in &v { // Se utiliza una referencia &v
      println!("{}", num);
   }
   println!("{}", v[0]); // v sigue disponible
}
```

En cambio, en el siguiente ejemplo, el bucle utiliza directamente la colección y consume la propiedad de la misma. Al finalizar el bucle, no se puede acceder a los elementos de la colección:

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];
    // Se recorre la colección, se pasa la la propiedad de v
    for num in v {
        println!("{}", num);
    }
    // Ahora v ya no está disponible
    println!("{}", v[0]); // Error
}
```

En Rust, al definir los iteradores, se hace referencia al tipo de elementos que recorre, pero no al tipo de colección o a la fuente de los datos que los alberga. Ésto permite definir funciones que describen la intención de lo que se quiere conseguir al procesar una fuente de datos, de manera independiente a la forma que adopten dichos datos. De esta forma, se podría cambiar la fuente de datos o su forma interna de organización, manteniendo las funciones que procesan dichos datos. Se trata de una codificación más declarativa que los bucles.

Los iteradores proporcionan numerosos métodos que permiten operar con los elementos que se iteran. Básicamente hay dos tipos de métodos:

- Métodos que devuelven un nuevo iterador obtenido al realizar alguna operación sobre los elementos del iterador original. Se suelen llamar *adaptadores* o también *iteradores internos*. Ejemplos de estos métodos serían *map()* y *filter()*.
- Métodos que realizan alguna operación directamente sobre los elementos del iterador. Se suelen denominar *iteradores externos*. Un ejemplo de estos métodos sería *for_each()*.

Los métodos adaptadores permiten encadenar operaciones haciendo que la salida de cada uno ellos, sea la entrada del siguiente. Ésto favorece la *canalización del procesamiento* de los datos (*processing pipeline*). La codificación obtenida es declarativa y muestra de manera explícita qué se quiere hacer con los datos, en contra de lo que sucede al realizar el procesamiento a base de bucles.

La totalidad de los métodos que proporcionan los iteradores se pueden consultar en la documentación de la librería estándar, en la siguiente dirección:

`https://doc.rust-lang.org/std/iter/trait.Iterator.html`

Una característica de los iteradores en Rust es que actúan de manera perezo-

sa, esto es, las operaciones asociadas a sus métodos no se ejecutan hasta que son necesarias. En concreto, es habitual tras una serie de operaciones encadenadas utilizando los métodos del iterador, querer convertir el iterador resultante en algún tipo de colección, como un vector u otra. Para ello, se suele utilizar el método *collect()*, que permite obtener una colección a partir de un iterador. En ese momento se ejecutarán las operaciones. En los ejemplos que siguen se verá cómo utilizar el método *collect()* en diferentes situaciones.

Muchos de los métodos que se utilizan cuando se trabaja con iteradores son más o menos estándar y están disponibles en todos los lenguajes que hacen uso de iteradores, si bien, en algunos casos pueden utilizar nombres diferentes.

Los métodos de los iteradores es frecuente que sean ejemplos paradigmáticos de funciones de orden superior: reciben una función como argumento que es la que se utilizará para operar sobre los elementos del iterador. También es frecuente que la función que se pasa como argumento sea una closure.

Tres de estos métodos que podríamos decir que son estándar en cualquier lenguaje que utilice iteradores son los métodos *map*, *filter* y *fold*:

- *map()*: este método recibe como argumento una función que se aplica sobre cada uno de los elementos del iterador. El método *map* devuelve un nuevo iterador cuyos elementos no tienen por qué ser del mismo tipo que los del iterador original.
- *filter()*: recibe como argumento una función que al aplicarla a los elementos del iterador devuelve *true* o *false*. A esta función que recibe como argumento se le llama el *filtro*. El método *filter* devuelve un nuevo iterador con los elementos del iterador original en los que el filtro devuelve *true*.
- *fold()*: este método acumula los valores del iterador en un valor que puede ser de otro tipo diferente que el de los elementos del iterador. También utiliza una función como argumento. Es posible encontrar este método en otros lenguajes con el nombre *reduce*. En Rust también hay un método *reduce*, similar a *fold()*.

En los siguientes apartados se van a tratar en profundidad estos tres métodos.

13.7 El método *map()*

Como se ha dicho, el método *map()* recibe como argumento una función y la aplica a los elementos del iterador para obtener un nuevo iterador. La función que se pasa como argumento a *map()*, toma como argumento un elemento del

iterador original y devuelve otro elemento, que puede ser del mismo tipo o de otro. El resultado final es un nuevo iterador con elementos que no tienen por qué ser del mismo tipo que los del iterador original. La Figura 13.1 muestra el esquema de funcionamiento de *map()*.

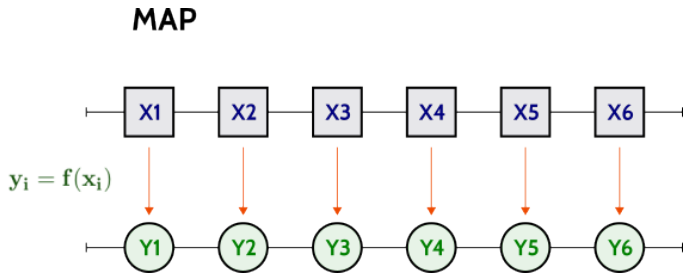

Figura 13.1: Esquema de funcionamiento de *map()*

En el siguiente código se crea un vector *v* de números enteros y se obtiene un nuevo vector formado por los cuadrados de dichos números.

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];

   let iterador_original = v.into_iter();
   let new_iterador = iterador_original.map(|x| x*x);
   let new_vector: Vec<i32> = new_iterador.collect();

   println!("{:?}", new_vector); // Imprime [1, 4, 9, 16, 25]
}
```

Observe los pasos que se han seguido:

1. Se obtiene un iterador a partir del vector *v* utilizando el método *into_iter()*.
2. Se obtiene un nuevo iterador aplicando el método *map* al iterador original. El método *map* recibe como argumento una closure que, para cada valor *x* del iterador original, devuelve su cuadrado.
3. El iterador resultante de la aplicación de *map()* se convierte de nuevo en un vector utilizando el método *collect()*. Al método *collect* hay que indicarle el tipo de colección que se quiere obtener, que en este caso es *Vec<i32>*.

Aunque en el ejemplo se ha desglosado cada paso para hacerlo más didáctico, lo habitual es hacer todas las operaciones encadenadas. También es habitual

ir poniendo las sucesivas operaciones en diferentes líneas, utilizando la tabuladores para dar claridad al código. El siguiente ejemplo es equivalente al anterior:

```
fn main() {
   let v = vec![1, 2, 3, 4, 5];

   let new_vector: Vec<i32> = v.into_iter()
   .map(|x| x*x)
   .collect();

   println!("{:?}", new_vector); // Imprime [1, 4, 9, 16, 25]
}
```

Como se ha dicho, el iterador resultante de la aplicación de *map()* puede tener elementos de distinto tipo que el iterador original. En el siguiente ejemplo, el iterador original es de cadenas de caracteres y se obtiene un iterador con el número de caracteres de cada una de esas cadenas:

```
fn main() {
   let v = vec!["Uno", "Dos", "Tres", "Cuatro", "Cinco"];

   let new_vector: Vec<i32> = v.into_iter()
   .map(|x| x.len() as i32)
   .collect();

   println!("{:?}", new_vector); // Imprime [3, 3, 4, 6, 5]
}
```

13.8 El método *filter()*

El método *filter()* recibe como argumento un función de retorno lógico: *true* o *false*. El parámetro de la función es del tipo de los elementos del iterador. El resultado es un nuevo iterador formado por los elementos del iterador original que *pasan el filtro*, esto es, los elementos en los que la función filtro devuelve *true*. La Figura 13.2 esquematiza el funcionamiento de *filter()*.

El siguiente ejemplo parte de un vector de tuplas con las de dos coordenadas *f64* de una serie de puntos. Se aplica un filtro para extraer los puntos del vector

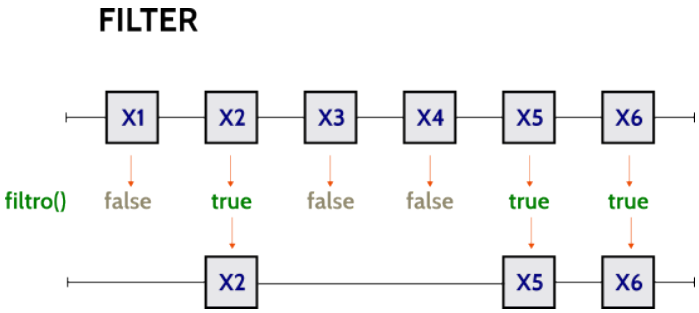

Figura 13.2: Esquema de funcionamiento de la función *filter()*

original que son interiores a un círculo de radio *1* centrado en el origen.

```
fn main() {
   let v: Vec<(f64, f64)> = vec![
   (1., 1.), (0.5, -0.25),
   (2., 0.1), (-0.5, -0.5)
   ];

   let new_vector: Vec<(f64, f64)> = v.into_iter()
   .filter(|(x, y)| (x * x + y * y).sqrt() < 1.0)
   .collect();

   println!("{:?}", new_vector); // [(0.5, -0.25), (-0.5, -0.5)]
}
```

Observe que en la función *filter*, a diferencia de lo que ocurría con la función *map()*, los elementos del iterador resultante son del mismo tipo que los del original. De hecho, son un subconjunto de los elementos del iterador original.

13.9 El método *fold()*

El método *fold()* aplica la función que recibe como argumento a los valores del iterador original, acumulando los valores en un solo valor que no tiene por qué ser del mismo tipo. A diferencia de *map* y *filter*, el método *fold()* tiene dos parámetros: el valor inicial del acumulador y la función que se utilizará para realizar los cálculos. La signatura de *fold()* es la siguiente:

```
fn fold(valor_inicial_acumulador: B, f)->B)
```

La función *f* es del tipo *FnMut*, recibe dos parámetros: el elemento del iterador

original que se está procesando y un valor del tipo de datos *B* del acumulado; la función debe devolver un valor del tipo de datos *B* del acumulado. Llamando *A* al tipo de datos de los valores del iterador original y *B* al tipo de datos del valor acumulado, esto se puede expresar así:

```
f: FnMut(B, T) ->B
```

Seguramente, un ejemplo ayude a comprender el funcionamiento de la función *fold()* mejor que tanta explicación. El siguiente código, calcula la suma de los elementos de un vector de números enteros:

```
fn main() {
   let v: Vec<i32> = vec![1, 2, 3, 4, 5];

   let suma: i32 = v.iter()
   .fold(0, |acc, x| acc+x);
   println!("{suma}"); // Imprime 15
}
```

Observe que, en este caso, el valor inicial del acumulador es *0*. La función acumuladora declara dos parámetros: *acc* y *x*. En cada iteración, *acc* contendrá el actual valor del acumulador. La función devuelve la suma de *acc* y *x*.

Observe también que, para obtener el iterador del vector, se ha utilizado el método *iter()*, por lo que las *x* de la función acumuladora son referencias a los valores del vector original y no se consume el vector.

La variable *acc* que se utiliza para el acumulador necesita ser mutable, de ahí que se haya dicho que la función acumuladora es del tipo *FnMut*.

En el siguiente ejemplo, los valores del iterador original son del tipo *char*, mientras que el valor acumulado es del tipo *String*:

```
fn main() {
   let v: Vec<char> = vec!['h', 'o', 'l', 'a'];

   let suma: String = v.into_iter()
   .fold(String::new(), |mut acc, c| {
      acc.push(c);
      acc
   }
   );
   println!("{suma}"); // Imprime hola
}
```

En este caso, ha sido necesario declarar que el parámetro *acc* de la closure es mutable. También ha sido necesario operar en dos pasos, pues la función *push()* modifica la cadena original pero devuelve el valor unitario *()*.

Otro ejemplo clásico es el cálculo del máximo de una serie de números:

```
fn main() {
   let v: Vec<i32> = vec![-1, 2, 5, 1, -2];

   let maximo: i32 = v.iter()
   .fold(v[0], |acc, x | if *x>acc {*x} else {acc});
   println!("{}", maximo); // Imprime 5
}
```

Un ejemplo un poco más complejo podría ser el calculo de la longitud de una polilínea definida mediante las coordenadas de sus puntos:

```
fn main() {
   let v: Vec<(f64, f64)> = vec![(0., 0.), (1., 1.), (2., 2.)];

   let acumulado: (f64, f64, f64) = v.iter()
   .fold((v[0].0, v[0].1, 0.0), |acc, p |{
      let d = ((p.0 - acc.0)*(p.0 - acc.0) +
      (p.1 -acc.1)*(p.1 -acc.1)).sqrt();
      (p.0, p.1, acc.2 + d)
   }
   );
   println!("{:.4}", acumulado.2); // Imprime 2.8284
}
```

Observe que, en este caso, cada iteración necesita las coordenadas del punto anterior, de ahí la técnica que se ha utilizado para definir los valores acumulados como una tupla de tres componentes.

13.9.1 Ejemplo: cálculo del máximo

13.10 Herramientas de los Iteradores

Rust ofrece otras funciones que se pueden aplicar a los iteradores.

- **for_each():** devuelve la lista resultante de aplicar la closure que recibe como argumento a cada elemento de la lista original. Es equivalente a un bucle *for*, aunque en determinadas situaciones puede ser más rápido.
- **filter():** devuelve la lista resultante de filtrar los elementos de la lista original utilizando para ello la closure que recibe como argumento. La closure acepta como parámetro un elemento del tipo de datos de los elementos de la lista y devuelve *true* o *false*. La lista resultante contendrá solo los elementos en los que la closure devuelva *true*.

  El siguiente ejemplo extrae los elementos pares del array original:

```
fn main() {
   let lista = [1, 2, 3, 4, 5];
   let nueva_lista: Vec<&i32> = lista.iter().filter(|&x| *x%2==0).
   println!("{:?}", nueva_lista); // [2, 4]
}
```

En el ejemplo anterior, la función *filter()* opera sobre un iterador de referencias a valores, por lo que primero hay que convertir el array original en un iterador de referencias a números enteros usando el método *iter()*. La closure recibe una referencia a un elemento como parámetro, *&x*, y comprueba si el resto de dividir por 2 el valor al que apunta dicha referencia es igual a 0. Sucede que, a priori, no se conoce el tamaño del array resultante y el tamaño de un array hay que conocerlo en tiempo de compilación, por lo que se utiliza el método *collect()* para convertir el iterador resultante en un vector de referencias a números enteros.

La totalidad de las funciones aplicables a los iteradores se pueden consultar en el siguiente enlace:

(Fuentes:

- `https://blog.jetbrains.com/rust/2024/03/12/rust-iterators-beyond-the`
- Module iter: `https://doc.rust-lang.org/std/iter/`
- Processing a Series of Items with Iterators: `https://doc.rust-lang.org/book/ch13-02-iterators.html`
- `https://mustafabugraavci.blog/2024/04/23/mastering-iterators-in-rust`

CAPÍTULO 14

Funciones puras, inmutabilidad

Contenido

14.1 Distinguir entre datos, cálculos y acciones
14.2 Tipos de entradas y salidas de las funciones
14.3 Separar las cosas
14.4 Extraer los cálculos de las acciones
14.5 Ejemplo: eliminar entradas implícitas
14.6 Inmutabilidad. Copy-on-write

Como se indicó en el Capítulo 1, una de las características distintivas de la Programación Funcional es su preferencia por la utilización de las funciones puras, esto es, funciones sin efectos secundarios. Algunos autores utilizan la denominación cálculos, para referirse a las funciones puras y acciones, para referirse a las funciones con efectos secundarios [11]. Esta será la denominación que se utilizará preferentemente en el texto a partir de este momento.

Otro de los pilares de la programación funcional es la utilización de las estructuras de datos inmutables, lo que proporciona ventajas a la hora de hacer determinadas optimizaciones en el código compilado y facilita la programación concurrente y en paralelo.

En este capítulo se va a ahondar en la utilización de las funciones puras y variables inmutables. Las técnicas que se van a explicar, que proceden en su mayoría de patrones de diseño de la Programación Funcional, no dependen del lenguaje de programación que se utilice y su aplicación produce beneficios al código generado, haciéndolo más fácil de comprender, favoreciendo su reutilización y facilitando el proceso de pruebas y depuración. Se pueden aplicar por igual a la programación orientada a objetos o a la programación basada en procedimientos. En realidad, se podría decir que se trata simplemente de buenas prácticas de programación.

Hay dos ideas principales que subyacen en todas estas técnicas:

- *Separar las acciones de los cálculos y de los datos.*
- *Propiciar el uso de abstracciones de orden superior.*

14.1 Distinguir entre datos, cálculos y acciones

El código de los programas se puede dividir en tres categorías:

- Datos.
- Cálculos (funciones puras, funciones sin efectos secundarios).
- Acciones (funciones impuras, funciones con efectos secundarios).

Los *datos* son la plasmación de hechos que han sucedido. Los datos no producen efectos secundarios ni dan lugar a ningún resultado. Son código inerte. Lo que sí admiten, son diferentes interpretaciones. Un mismo dato, por ejemplo una cadena de texto con una palabra, puede admitir diferentes interpretaciones según el programa concreto que haga uso de ella.

Las *funciones puras* son aquellas que no producen efectos secundarios, no dependen de ningún estado exterior y siempre proporcionan el mismo resultado para el mismo valor de los argumentos de entrada (ver Figura 14.1). Se suelen denominar también *funciones matemáticas* o *cálculos*. En este texto, siguiendo la denominación utilizada por Eric Normand, se denominarán frecuentemente *cálculos* [11].

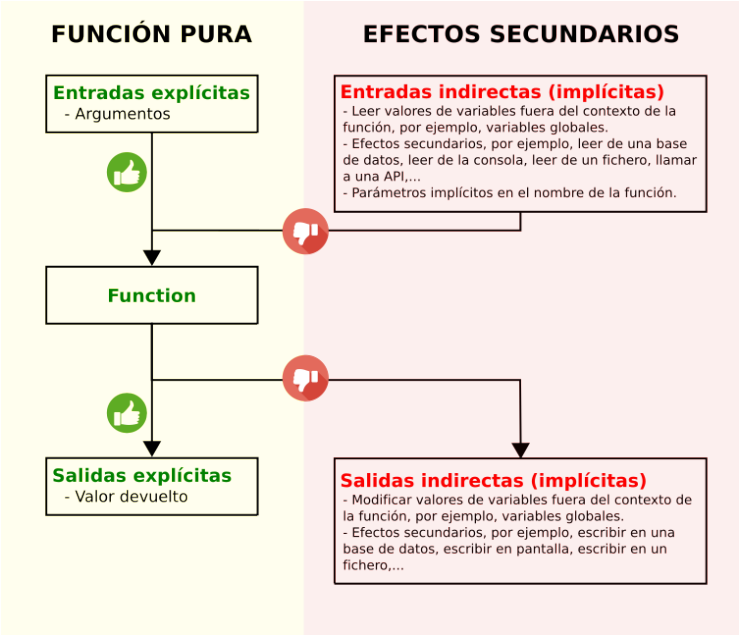

Figura 14.1: Funciones puras y efectos secundarios.

Se entiende por *efecto secundario* cualquier acción del programa que no sea un cálculo. Por ejemplo, modificar el estado de una variable, mostrar un mensaje en una pantalla o enviar un correo electrónico serían ejemplos de efectos secundarios. Cuando una función produce algún efecto secundario, se dice que es una *función impura*, una *función con efectos secundarios* o, siguiendo la denominación de Eric Normand, una *acción*. Así se denominarán frecuentemente en el texto a partir de ahora.

No es posible hacer un programa de utilidad sin producir efectos secundarios. De hecho, los efectos secundarios suelen ser la razón de ser de los programas. Lo que sí se puede hacer es codificar los programas de forma que las acciones siempre se lleven a cabo dentro de ámbitos controlados y bajo una supervisión minuciosa.

Al inspeccionar cualquier fragmento de código, hay que distinguir las partes que son acciones, de las que son cálculos y de las que son simplemente datos. En general, se debe dar preferencia a la utilización de datos sobre los cálculos y a los cálculos sobre las acciones.

Un ejemplo de cálculo sería la función siguiente:

```
fn cuadrado(x: f64) -> f64 {
   x*x
}
```

La función anterior, recibe como argumento un número del tipo *f64* y devuelve el cuadrado de dicho número. El código dentro de la función no afecta a nada externo a la misma, no tiene ninguna entrada de información que no sea a través de su parámetro ni ninguna salida que no sea a través de su valor devuelto; el valor que devuelve, siempre será el mismo si recibe el mismo valor como argumento, independientemente de en qué momento se llame a la función o cuántas veces se llame a la misma.

Las acciones, en cambio, son funciones con efectos secundarios. Los efectos secundarios pueden tomar distintas formas dentro del código. Por ejemplo:

- Llamadas a otras funciones o métodos: `println!("Hola, qué tal")`
- Constructores: `new Date()`
- Expresiones: `x` (si `x` es mutable).
- Declaraciones: `x = 3` (si se muta `x`).

Las funciones que son acciones, pueden estar compuestas por otras acciones, cálculos y datos. Una función que sea un cálculo puede estar compuesta por varios cálculos menores. Los datos solo pueden contener otros datos.

A continuación, se van a explicar con más detalle las características de los datos, los cálculos y las acciones.

14.1.1 Datos

Los datos son hechos resultantes de acontecimientos que han sucedido a lo largo de la ejecución del programa. Sus características principales son las siguientes:

- Se implementan utilizando los tipos de datos que ofrezca el lenguaje de programación que se esté utilizando.
- Las estructura de los datos encierra parte de su significado, por ejemplo, el orden de los elementos en una lista.
- Los datos en sí mismos, son inmutables. Los datos de un acontecimiento no varían, lo que puede suceder es que se produzcan nuevos acontecimientos que den lugar a nuevos datos.

- *Ventajas de los datos:* son serializables, se pueden comparar para comprobar su igualdad, están abiertos a distintas interpretaciones.
- *Desventajas:* el hecho de que admitan distintas interpretaciones puede ser un arma de doble filo. Un cálculo es útil, aunque no entendamos cómo está hecho, pero un dato no tiene ningún significado sin una interpretación adecuada.

En los programas, conviene separar la generación de los datos de su utilización. Por ejemplo, si un programa tiene que enviar unos correos a una serie de personas, conviene proceder primero a generar los correos, guardarlos en una estructura de datos y luego enviarlos.

Es preferible hacerlo así, que ir generando los correos uno a uno e ir enviando cada correo que se genera. Esto, facilitará la realización de test: es más fácil probar una función que genera una lista de datos que una función que envía un correo.

Si la lista de correos fuera muy grande, siempre se podría repetir el procedimiento anterior dividiendo en subgrupos los correos que se tienen que enviar.

14.1.2 Cálculos: funciones puras

Los cálculos son las funciones puras, también llamadas funciones matemáticas. A partir de una entrada se genera una salida. En una función que sea un cálculo, la entrada se produce a través de los parámetros de la función, la salida es el valor que devuelve y el cálculo es el cuerpo de la función.

Los cálculos gozan de *trasparencia referencial*, esto es, en cualquier lugar del código en el que aparece la llamada a una función que es un cálculo, se puede sustituir dicha llamada por el resultado que se obtiene. Cuando el compilador identifica un cálculo, puede proceder a optimizaciones sustituyendo la llamada a la función por su resultado. Esto no es posible garantizarlo en el caso de que la función sea una acción.

Las características de los cálculos:

- En los programas, los cálculos se implementan utilizando funciones.
- El significado que encierran es el cálculo que hacen.
- Las ventajas de los cálculos frente a las acciones son:
    - Es más fácil hacer test y probarlos.
    - Los test se pueden automatizar más fácilmente.
    - Es posible componer unos cálculos con otros.
    - No hay que preocuparse de otros cálculos que se pudieran estar pro-

duciendo al mismo tiempo, el resultado solo depende del valor de los argumentos de entrada (programación en paralelo).
    - Por el mismo motivo, no hay que preocuparse de lo que haya sucedido antes de llamar al cálculo o de lo que pueda suceder después.
    - No hay que preocuparse de cuántas veces se haya llamado anteriormente al cálculo.
- *Desventajas:* al igual que sucede con las acciones, no se puede estar seguro de lo que hacen si no se ejecuta el código. Lo único que se está seguro de lo que hacen son los datos, que no hacen nada.

14.1.3 Acciones: funciones con efectos secundarios

Las funciones que tienen entradas o salidas implícitas, son *acciones*, también llamadas *funciones con efectos secundarios* o *funciones impuras*. Una acción es cualquier cosa que tenga un efecto en el mundo exterior a la función o que sea afectada por el mundo exterior.

Sus características principales son:

- Las acciones se implementan en el código utilizando funciones.
- Se propagan por el código: si una función llama a otra función que es una acción, ella misma se convierte en una acción.
- Las acciones dependen o pueden depender de cuándo se ejecutan (del orden de ejecución) o de cuantas veces se ejecutan (repetición).
- El significado de las acciones es el efecto que producen en el mundo exterior. Hay que asegurarse de que producen el efecto deseado.
- Las acciones son inevitables. De hecho, son la razón de ser de los programas.

La Programación Funcional propone diversas técnicas para gestionar de manera adecuada las acciones:

- Reducir el número de acciones a las estrictamente necesarias.
- Mantenerlas reducidas a su mínima expresión, extrayendo de ellas todos los elementos que sean cálculos o datos.
- Restringir las acciones a las interacciones con el exterior. En el interior, idealmente, todo son cálculos y datos.
- Reducir la dependencia del tiempo. Hacer que las acciones dependan menos de cuándo se jecutan o de cuántas veces se ejecutan.

14.2 Tipos de entradas y salidas de las funciones

Todas las funciones tienen entradas y/o salidas. Se pueden clasificar en implícitas y explícitas:

- *Entradas explícitas*: a través de los parámetros.
- *Salidas explícitas*: a través del valor devuelto.
- *Entradas o salidas implícitas*: cualquier otra forma de entrada o salida de información de la función.

Las entradas *explícitas* son los valores de los argumentos que recibe la función. Cualquier otra entrada de información a la función se considera una entrada *implícita*. Las salidas *explícitas* son los valores devueltos. Cualquier otra salida de información de la función se considera una salida *implícita* (ver Figura 14.1).

La existencia de entradas o salidas implícitas convierten una función en impura. Una *entrada implícita* puede ser leer el valor de una variable externa a la función, por ejemplo una variable global. También sería un entrada implícita la consulta de datos provenientes de una base de datos. Las *salidas implícitas* son cualquier información que salga de la función y que no sea el valor devuelto, por ejemplo, modificar el valor de una variable externa o enviar un email o escribir informacion en una base de datos o un fichero.

Cualquier entrada o salida implícita de información a la función convierte la función en impura, en una acción.

Si en una función se eliminan todas las entradas y salidas implícitas, se convierte en un cálculo. Las entradas implícitas, hay que convertirlas en parámetros y las salidas implícitas hay que convertirlas en valores devueltos.

14.3 Separar las cosas

Conviene recordar el concepto de *refactorizar*. Consiste en hacer modificaciones en el código organizándolo de otra manera sin alterar su comportamiento. Hay diversas técnicas de refactorización que se han estandarizado, por ejemplo, la denominada *extraer subrutina* consiste en extraer una parte de una función a otra función independiente.

En general, la refactorización da lugar a un código con más líneas, pero es más comprensible, más reusable y es más fácil hacer los test.

Una técnica de diseño reconocida consiste en *separar las cosas* (*pull things apart*). Ésta técnica también es conocida como el *Principio de Responsabilidad única*: que cada función haga solo una cosa y que la haga bien. Da lugar a un

código con más funciones y más pequeñas, pero facilita la reutilización y hace el código más fácil de comprender. En la Programación Funcional se promueve la utilización de funciones más pequeñas, separando las cosas. Los procesos complejos se consiguen mediante la *composición* de dichas funciones. La figura 14.2 trata de esquematizar está técnica.

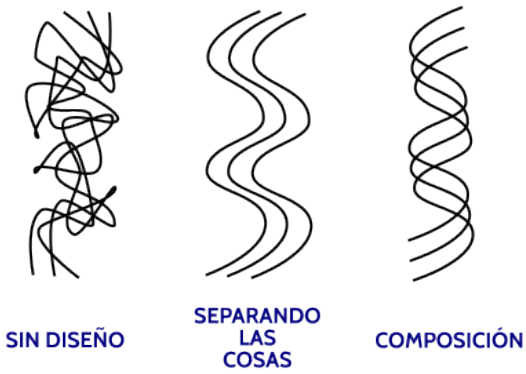

Figura 14.2: Visualización de la separación de diferentes cosas y su composición

14.4 Extraer los cálculos de las acciones

Observe la función *grabar_caudal()* que se muestra en el siguiente fragmento de código:

```
fn grabar_caudal() {
   let velocidad = leer_sensor();
   let caudal = 1.8 * velocidad.powf(3.2);
   save_caudal_to_database(caudal);
}
```

La función lee el valor de la velocidad de un sensor, realiza el cálculo del caudal asociado a esa velocidad y graba el resultado en una base de datos. La función es una acción: no tiene ningún argumento de entrada ni devuelve ningún resultado, pero tiene una entrada de datos implícita a través de la función *leer_sensor()* y una salida de datos implícita que graba datos en la base de datos utilizando la función *save_caudal_to_database()*.

Además, la función incluye un cálculo, que aquí se ha expresado mediante una operación relativamente sencilla, pero que podría ser más complicado.

Si se quisiera hacer algún test para comprobar que el cálculo se realiza de manera correcta, no sería fácil. Una opción sería habilitar el sensor correspondiente y la base de datos asociada mientras se hacen los test. Para comprobar que el resultado calculado es correcto, seguramente habría que consultar el valor grabado en la base de datos, pues la función no devuelve ningún valor. Si se quisiera probar el cálculo con diferentes valores de velocidad, habría que *trucar* el sensor, lo que puede resultar complicado.

El problema está en que se están mezclando dos cosas: una acción (en ralidad, dos) y un cálculo. La solución adecuada es extraer el cálculo a una función independiente, como se muestra en el siguiente código:

```
fn grabar_caudal() {
   let velocidad = leer_sensor();

   let caudal = calcula_caudal(velocidad);

   save_caudal_to_database(caudal);
}
fn calcula_caudal(velocidad: f64) -> f64 {
   let resultado = 1.8 * velocidad.powf(3.2);
   resultado
}
```

Ahora hay dos funciones: una acción y un cálculo. Es fácil realizar los test para comprobar que los cálculos se hacen de manera adecuada. Se puede probar la función *calcula_caudal()* para cualquier valor de entrada de manera sencilla.

Los pasos que se han seguido se podrían esquematizar de la siguiente forma:

- En la función original, seleccionar el fragmento de código a extraer.
- Ponerlo en una nueva función, sustituyéndolo en la función original por una llamada a la función recién creada. Es posible que haya que añadir algún parámetro.
- Identificar todas las entradas y salidas en la nueva función, convirtiendo las entradas en parámetros y las salidas en valores devueltos, que habrá que asignar a alguna variable en la función original.
- Los argumentos y lo valores devueltos de la función recién creadas tienen que ser inmutables, de forma que la nueva función sea un cálculo.

14.5 Ejemplo: eliminar entradas implícitas

En numerosas ocasiones, una misma función hace más de una cosa. Suponga que se tiene la siguiente función:

```
fn main() {
   facturar(100.0);
}
fn facturar(precio: f64) {
   let descuento = 0.10 * precio;
   println!("Precio      : {:.2}", precio);
   println!("Descuento   : {:.2}", descuento);
   println!("Precio neto : {:.2}", precio - descuento);
}
```

La función *facturar()* recibe como argumento el precio de un artículo, calcula el descuento a aplicar y muestra el resumen de la factura en pantalla. Está haciendo dos cosas: calcular el descuento y mostrar los resultados en pantalla. El hecho de mostrar los resultados en pantalla es un efecto secundario, una salida implícita que convierte a la función en una acción. Pero, además, tiene una entrada implícita de información. El valor 0.10 del porcentaje de descuento que se aplica es una especie de variable global, correspondiente a la lógica de negocio de la aplicación.

No es posible eliminar la acción, de hecho, es el objetivo del programa: mostrar la factura en pantalla, una vez que se ha aplicado el descuento. Es difícil diseñar un test para probar la función. Por una parte hay que probar que los cálculos son correctos y, por otra, que la salida de la función también es correcta. Tenga en cuenta que, aunque la salida se ha planteado simplemente mostrando unos mensajes en pantalla, podría tratarse de la actualización de ciertos registros en una base de datos o cualquier otra salida. En el caso de hacer la salida a una base de datos, para probar los cálculos, habría que habilitar y tener disponible la base de datos.

Lo correcto es separar el cálculo del descuento de la salida de resultados. Pero, además, la entrada implícita del porcentaje de descuento hay que convertirla en un parámetro de la función que calcula el descuento. Podría hacerse de la siguiente forma:

```
fn main() {
   let porcentaje_descuento = 0.10;
   let precio = 100.0;
   let descuento =
   calcula_descuento(precio, porcentaje_descuento);
   facturar(precio, descuento);
}
fn calcula_descuento(precio: f64, porcentaje: f64) -> f64 {
   precio*porcentaje
}
fn facturar(precio: f64, descuento: f64) {
   println!("Precio      : {:.2}", precio);
   println!("Descuento   : {:.2}", descuento);
   println!("Precio neto : {:.2}", precio - descuento);
}
```

Ahora, la función *calcula_descuento()* es una función pura, un cálculo. Sus únicas entradas se producen a través de los argumentos que recibe y su salida es el valor que devuelve. No importa cuándo se llame a la función o cuántas veces se la llame: si recibe los mismos argumentos siempre devolverá el mismo resultado. Es fácil plantear los test para probar que realiza los cálculos de manera adecuada.

Además, el porcentaje del descuento que se tiene que aplicar se ha independizado de cualquiera de las funciones. Está centralizado en una variable accesible para otras funciones del programa. Si la empresa decidiera cambiar su política de descuentos, solo tendría que modificar el valor de la variable en un sitio y se actualizaría la política en cualquier función que haga uso de ella.

Por otra parte, la función *facturar()* sigue siendo una acción, pero independiente de otras consideraciones. Si se decidiera cambiar la forma de salida de la pantalla a una base de datos, por ejemplo, solo habría que actuar en dicha función, despreocupándose del cálculo de los valores asociados. Aun siendo una acción, se han eliminado sus entradas implícitas, convirtiéndolas en parámetros.

En el ejemplo, se han realizado varias refactorizaciones:

- Extraer un cálculo de una acción.
- Convertir una entrada implícita en un parámetro.
- Separar las cosas en funciones independientes.

14.6 Inmutabilidad. Copy-on-write

Al inicializar una variable, se cambia su valor. Si en las funciones solo se inicializan variables locales, nada fuera de la función ve dichos cambios, salvo que lo vean a través del valor devuelto.

Ejemplo sin copy-on-write:

```
fn main() {
   let mut list = vec![1, 2, 3];
   list.push(4);
   assert_eq!(list, vec![1, 2, 3, 4]);
}
```

Sin copy-on-write, pero con función:

```
fn main() {
   let mut list = vec![1, 2, 3];
   modify_list(&mut list, 4);
   assert_eq!(list, vec![1, 2, 3, 4]);
}
fn modify_list(list: &mut Vec<i32>, x: i32) {
   list.push(x);
}
```

Con copy-on-write:

```
fn main() {
   let list = vec![1, 2, 3];
   let list = copy_on_write(list, 4);
   assert_eq!(list, vec![1, 2, 3, 4]);
}
fn copy_on_write(list: Vec<i32>, x: i32) -> Vec<i32> {
   let mut new_list = list;
   new_list.push(x);
   new_list
}
```

No se modifica la lista original, se transforma en una nueva lista que incorpora las modificaciones. No hay pérdida de rendimiento, la propiedad de los valores se va trasmitiendo, sin modificar su posición en memoria.

Variante usando un *RefCell*:

```
fn main() {
   let list = RefCell::new(vec![1, 2, 3]);
   let list = modify_cell(list, 4);
   assert_eq!(list.take(), vec![1, 2, 3, 4]);
}
fn modify_cell(list: RefCell<Vec<i32>>, x: i32)
-> RefCell<Vec<i32>> {
   let new_list = list;
   new_list.borrow_mut().push(x);
   new_list
}
```

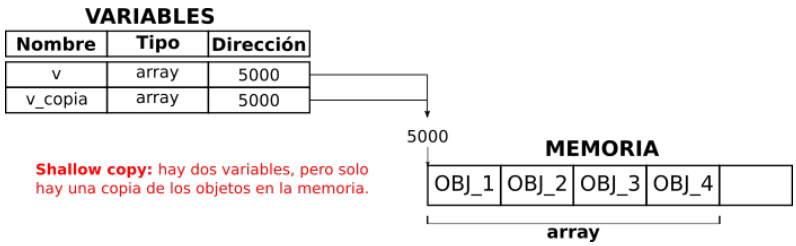

Figura 14.3: Esquema del mecanismo *shallow-copy* de un *array* en Java: se copian las referencias a los objetos, no los objetos en sí mismos. Al final hay dos variables apuntando a los mismos objetos en memoria.

CAPÍTULO 15

Colecciones

CAPÍTULO 16

Patrones de diseño en la Programación Funcional

CAPÍTULO 17

Grokking Simplicity

Contenido

17.1 Welcome to Grokking simplicity
17.2 Functional thinking in action
17.3 Distinguishing actions, calculations and data
17.4 Distinción entre acciones, cálculos y datos
17.5 Ejemplo
17.6 Extracting calculations from actions
17.7 Improving the design of actions
17.8 Staying inmutable in a mutable language
17.9 Staying inmutable with untrusted code
17.10 Stratified design (I)
17.11 Stratified design (II)
17.12 First-class functions (I)
17.13 First-class functions (II)
17.14 Functional iteration
17.15 Chaining functional tools
17.16 Functional tools for nested data
17.17 Isolating timelines
17.18 Sharing resources between timelines
17.19 Coordinating timelines
17.20 Reactive and onion architectures
17.21 The functional journey ahead

Apuntes del Libro «Grokking Simplicity», de Eric Norman [11]

17.1 Welcome to Grokking simplicity

17.2 Functional thinking in action

17.3 Distinguishing actions, calculations and data

Los programadores funcionales clasifican cualquier fragmento de código como acción, cálculo o datos. Esta clasificación puede recibir otras denominaciones, pero el concepto es el mismo.

- **Acciones:** son todo aquello que dependa de en qué momento se ejecuta o de cuántas veces se ejecuta. Por ejemplo, enviar un correo es una acción.
- **Cálculos:** son las tareas que solo dependen de los valores de entrada para generar un resultado. Los cálculos siempre devuelven el mismo resultado, si los valores de entrada son los mismos. Además, los cálculos nunca afectan a nada que esté fuera de ellos. Esto hace que los cálculos sean fáciles de testear y que su uso sea seguro, pues no hay que preocuparse de cuántas veces se utilicen o en qué orden sean invocados: si los parámetros de entrada son los mismos, el resultado será siempre el mismo.
- **Datos:** los datos son información registrada acerca de los acontecimientos. Tienen propiedades conocidas. Un mismo dato se puede interpretar de manera diferente según el contexto de ejecución en el que se encuadra. Por ejemplo, la factura de una cena la puede utilizar el cliente para llevar la cuenta de sus gastos mensuales o la puede utilizar el propietario del restaurante para determinar los gustos favoritos de sus clientes.

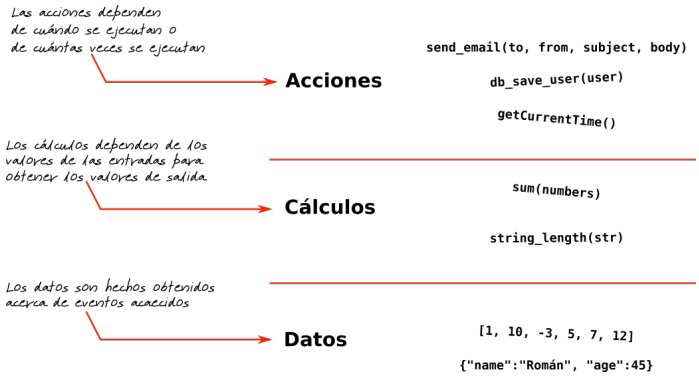

Figura 17.1: Ejemplos de código: Acciones, Cálculos y Datos

La proliferación de los sistemas distribuidos en los que intervienen diferentes dispositivos y peticiones cuasi simultáneas de información a las bases de datos,

el problema de organizar de manera adecuada el código se hace indispensable. En general, el código de las acciones será más difícil de comprender y de testear que el código de los cálculos, ya que estos últimos no dependen del número de veces que se invocan o del orden de dichas invocaciones. Es por ello que se hace importante separar en la medida de los posible el código que corresponde a las acciones del código que corresponde a los cálculos.

La programación funcional proporciona herramientas para el correcto tratamiento de cada una de estas categorías de código. En el caso de las acciones, es importante gestionar la forma en que cambia el estado de las variables del programa a lo largo del tiempo, garantizando el número de veces y el orden en el que se realiza cada acción. Los cálculos tienen su propia estrategia para comprobar su buen funcionamiento, a veces basada en técnicas matemáticas. En el caso de los datos, es importante organizarlos en estructuras que faciliten un acceso eficiente a la información que se quiere extraer de los mismos.

Hay dos técnicas fundamentales que permiten abordar los programas con un enfoque funcional:

- Distinguir en el código las acciones, de los cálculos y los datos.
- Utilizar abstracciones de primera clase.

A lo largo del curso se tratará de trasmitir el razonamiento funcional a la hora de abordar un problema de codificación. El objetivo es que las técnicas que se aprendan sean independientes del lenguaje que se utilice para programar y que sean de aplicación inmediata, tanto para la realización de un programa nuevo, como para refactorizar partes de un código ya existente.

Para clasificar el código en acciones, cálculos y datos es útil seguir los principios del *diseño estratificado*, separando el código en diferentes capas. Para organizar el orden en el que se ejecutan las acciones son de utilidad los *diagramas de tiempos* y la utilización de funciones de *primera clase*, que permiten utilizar otras funciones como parámetros o resultados.

En el diseño estratificado, se organiza el código en diferentes capas, ordenadas en función de la mayor o menor probabilidad de cambios en el código correspondiente a lo largo de la vida útil de la aplicación. Se suelen considerar tres capas principales:

- **Nivel técnico:** correspondería a la parte de la aplicación que tiene menos probabilidades de cambiar. Por ejemplo, el lenguaje de programación de la aplicación o las estructuras de datos que se utilizarán para almacenar la información.

- **Reglas del dominio de la aplicación:** si por ejemplo se está haciendo una aplicación para gestionar unos cultivos de hortalizas, las distancia optima a la que hay que poner las plantas en el terreno o la cantidad de humedad que necesitan corresponden al campo de conocimiento de dicho dominio técnico y es difícil que cambien durante la vida útil de la aplicación.
- **Reglas del negocio:** se consideran aquí reglas que vienen marcadas por la aplicación concreta que se esté desarrollando. En el ejemplo de los cultivos podrían ser los precios de los factores de producción o la disponibilidad de determinados recursos.

Cada capa de la aplicación se desarrolla sobre las demás y solo debe depender de las capas que hay situadas por debajo de ella. De esta forma, si se produce una modificación en algún elemento, se sabe que solo puede afectar a los elementos que estén situados en la misma capa o en las capas superiores. La Figura 17.2 muestra un ejemplo de organización en capas de una aplicación para gestionar cultivos.

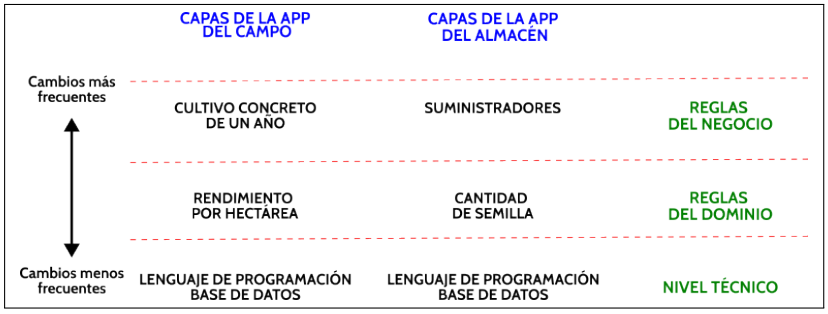

Figura 17.2: Esquema de capas en el diseño estratificado

17.3.1 Líneas de tiempos

En todas las aplicaciones hay que establecer el orden en el que se tienen que realizar las acciones. En aplicaciones sencillas, una distribución secuencial de las acciones puede ser suficiente. Pero, en sistemas distribuidos, en los que distintas tareas pueden correr a cargo de distintos componentes que pueden trabajar en paralelo o de forma concurrente, es importante establecer el orden en el que se tienen que realizar todas las acciones y los puntos en los que determinadas acciones no pueden ejecutarse si antes no se ha finalizado determinada acción anterior. Hay que cortar la línea de tiempos en algunos puntos para indicar que la ejecución no puede continuar hasta que se completen todas las tareas anteriores.

La Figura 17.3 muestra la línea de tiempos de una aplicación en la que, para poder ejecutarse las acciones *D* y *E* es necesario que primero se hayan completado las tareas llevadas a cabo por los componentes *A*, *B* y *C*.

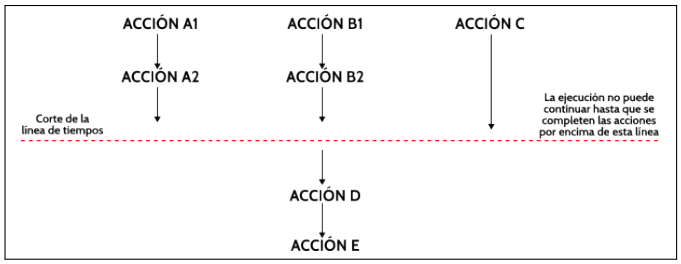

Figura 17.3: Cortando la línea de tiempo para garantizar el orden de ejecución de las acciones

La línea de tiempos ayuda a coordinar distintas acciones y a determinar los puntos en los que se pueden producir cuellos de botella durante la ejecución de los programas en sistemas distribuidos.

17.4 Distinción entre acciones, cálculos y datos

Antes de ponerse a codificar una aplicación nueva, conviene razonar .para distinguir qué elementos del programa serán acciones, cálculos o datos. En general, el orden de implementación consistirá en definir primero los datos, luego los cálculos y, por último, las acciones.

Cuando se está analizando un código ya existente, habrá que identificar igualmente las funciones que incluyen acciones. En muchos casos, una misma función incluirá acciones y cálculos. En esos casos, conviene refactorizar para separar las acciones de los cálculos.

Como se ha comentado, las acciones dependen de cuándo se ejecutan o de cuantas veces se ejecutan. Las funciones que implementan acciones se suelen denominar *funciones impuras* o *funciones con efectos secundarios*. Enviar un correo o leer datos de una base de datos serían ejemplos de acciones.

Por el contrario, los cálculos solo dependen del valor de los parámetros de entrada y no producen efectos secundarios. Se denominan *funciones puras* o *funciones matemáticas*. Calcular el máximo de una serie de números o comprobar si determinada dirección de correo es válida podrían ser ejemplos de funciones puras.

Se dice que los cálculos son *referencialmente trasparentes*. La trasparencia referencial significa que se puede sustituir el cálculo por el resultado sin que el

programa se vea afectado. Por ejemplo, si una función realiza la suma de dos números, en los lugares del programa donde se hace la llamada a la función se puede sustituir ésta por el resultado de la suma y el programa no se verá afectado.

Los datos son hechos.

17.4.1 Entradas y salidas implícitas

Los parámetros de una función son el procedimiento de *entrada explícita* de datos a la función. El valor devuelto por la función es la salida *explícita*. Cuando una función es impura, tiene entradas o salidas implícitas. Se denomina *entrada implícita* a la entrada de datos a la función que no procede de un parámetro. Se denomina *salida implícita* al valor que sale de la función sin hacerlo a través del valor devuelto por la misma.

El código del ejemplo muestra una función denominada *contador_impuro()* que no tiene parámetros ni devuelve ningún valor. En cambio, la función recibe como entrada implícita el valor de un contador existente en la base de datos, a través de un método llamado *get_contador_from_database()*, y realiza una salida también implícita reescribiendo en la base de datos el valor incrementado de dicho contador a través del método *update_contador_in_database()*. La salida implícita de la función es, además, un efecto secundario de la misma, convirtiendo a dicha función en una *acción*.

Ejemplo 17.1 Entradas y salidas implícitas

```
fn contador_impuro() {
   let contador = get_contador_from_database(); // Entrada implícita
   update_contador_in_database(contador+1); // Salida implícita
}
```

El acceso a variables globales dentro de una función es una forma de entrada o salida implícita.

En la medida de lo posible, hay que evitar las entradas y salidas implícitas, pues complican la trazabilidad y la facilidad de testeo de las funciones. En muchas ocasiones, las entradas implícitas se pueden sustituir por parámetros de la función. De la misma forma, las salidas implícitas es posible sustituirlas por valores devueltos por las funciones.

17.5 Ejemplo

Ejemplo 17.2 Ejemplo carrito

```
struct Producto {
   name: String,
   precio: f64,
}
fn main() {
   let mut carrito = Vec::<Producto>::new();
   let mut total_compra: f64 = 0.0;
   let producto = Producto{name: "Sandalias".to_string(), precio: 12.5
   add_producto(producto, &mut carrito, &mut total_compra);
}
fn add_producto(producto: Producto, carrito: &mut Vec<Producto>, total
   carrito.push(producto);
   calc_total_carrito(carrito, total_compra);
}
fn calc_total_carrito(carrito: &mut Carrito, total_compra: &mut f64) {
   *total_compra = 0.0;
   for producto in &carrito.items {
      *total_compra = *total_compra + producto.precio;
   }
   actualiza_web_total_compra( *total_compra);
}
fn actualiza_web_total_compra(total_compra: f64) {  }
```

17.6 Extracting calculations from actions
17.7 Improving the design of actions
17.8 Staying inmutable in a mutable language
17.9 Staying inmutable with untrusted code
17.10 Stratified design (I)
17.11 Stratified design (II)
17.12 First-class functions (I)
17.13 First-class functions (II)
17.14 Functional iteration
17.15 Chaining functional tools
17.16 Functional tools for nested data
17.17 Isolating timelines
17.18 Sharing resources between timelines
17.19 Coordinating timelines
17.20 Reactive and onion architectures
17.21 The functional journey ahead

CAPÍTULO 18

Functional Programming made easier (Scalfani)

18.1 1.- Discipline is freedom

18.1.1 Global State

El uso de varibles globales conlleva determinados inconvenientes:

- Cualquiera desde cualquier módulo puede cambiar el valor de las variables globales.
- Se producen acoplamientos de las variables globales entre sí y de unos módulos con otros.
- En programación concurrente no hay garantías respecto de la modificación de las variables.
- Colisión de nombres entre las variables globales y otros identificadores en cualquier módulo.

En el caso de la programación orientada a objetos se produce la misma circunstancia con el patrón *Singleton*.

Los lenguajes de PF prohíben la existencia de variables globales. Rust permite solo la existencia de constantes globales.

18.1.2 Mutable State

La posibilidad de que las variables puedan cambiar de valor también tiene algunos inconvenientes. Por ejemplo, es más difícil razonar sobre el código, pues los valores que pueden cambiar pueden hacer cambiar también la semántica del programa, o hacer el código más frágil.

En los lenguajes funcionales, las variables son inmutables y ello da lugar a algunas consecuencias.

- Las expresiones del tipo $x = x + 1$ no tienen sentido. Una vez que se asigna un valor a x, no se puede cambiar.

- Las asignaciones como $x = 20$ son *expresiones referencialmente trasparentes*, esto es, en cualquier parte del programa se puede utilizar indistintamente x o 20 con la seguridad de que el programa seguirá funcionando igual. La trasparencia referencial permite una evaluación *lazy* de la sustitución de x por su valor en el código.

Si las variables no pueden cambiar de estado, no es posible hacer bucles. En los lenguajes funcionales, los bucles se sustituyen por la recursividad. Cualquier bucle se puede ejecutar mediante recursividad y viceversa.

Por ejemplo, la definición matemática del factorial de un número se podría hacer de la siguiente forma:

$$n! = 1.2...(n-2).(n-1).n \quad \forall n \in \mathbb{N} \tag{18.1}$$

Con esta definición, parecería inmediato resolver el problema con un bucle, como se hace en el Ejemplo 18.1.

Ejemplo 18.1 Factorial calculado con un bucle

```
fn factorial_bucle(n: u32) -> u32 {
   let mut prod = 1;
   for i in 1..=n {
      prod = prod*i;
   }
   prod
}
```

Si en la Expresión 18.1 se cambia el orden del producto, los términos se podrían agrupar de la siguiente manera:

$$n! = n.(n-1)...2.1 \quad = n.(n-1)! \quad \forall n \in \mathbb{N} \tag{18.2}$$

Con esta definición surge el problema de calcular $0!$, pero su valor se puede deducir. De la Expresión 18.1 se sabe que $!1 = 1$. Se podría operar de la siguiente forma:

$$\begin{aligned} 1! &= 1 \\ 1! &= 1.(1-1)! = 1.0! \\ 1 &= 1.0! = 0! \end{aligned}$$

Con lo que finalmente queda que:

$$0! = 1 \tag{18.3}$$

Una vez calculado el valor de $0!$, se puede proceder a definir el factorial de cualquier número natural con la siguiente definición recursiva:

$$\begin{aligned} 0! &= 1 \\ n! &= n.(n-1)! \quad \forall n \in \mathbb{N} \end{aligned} \tag{18.4}$$

Esta definición de factorial se podría codificar como se hace en el Ejemplo 18.2.

Ejemplo 18.2 Factorial calculado de manera recursiva

```
fn factorial_recursivo(n: u32) -> u32 {
   match n {
      0 => 1,
      _ => n*factorial_recursivo(n-1)
   }
}
```

18.2 Variables globales

Vamos a ver en primer lugar cómo no se pueden usar las variables globales en Rust. Podríamos estar tentados de usar sentencias *let*, como en la variables locales de cualquier función:

```
use chrono::Utc;

let START_TIME: String = Utc::now().to_string();

fn main() {
   ...
}
```

El código anterior no compila, no se pueden usar asignaciones *let* en el ámbito global, pues la instrucción *let* crea una variable en la memoria stack, que no se inicializa hasta que se ejecuta el programa. Las variables globales solo se

pueden crear utilizando las cláusulas *const* o *static*, que utilizan la memoria del segmento de datos (*data segment*) del programa.

Tampoco se puede compilar la siguiente inicialización:

```
use chrono::Utc;

static START_TIME: String = Utc::now().to_string();

fn main() {
   ...
}
```

El compilador avisa que no se pueden utilizar funciones no constantes en la inicialización de variables globales. No se puede ejecutar ningún tipo de código antes de que el programa comience. El valor de una variable global debe ser conocido en el momento de la compilación, antes de la ejecución.

Tampoco valdría declarar la variable global y tratar de asignarle valor en *main()*:

```
use chrono::Utc;

static START_TIME: String;

pub fn main() {
   START_TIME =  = Utc::now().to_string();
   println!("{}", START_TIME);
}
```

El código anterior tampoco compila, el compilador nos avisa de que hay que asignar algún valor a la variable *static*. Podríamos pensar en asignarle un valor *None*

```
use chrono::Utc;

static mut START_TIME: Option<String> = None;

pub fn main() {
   START_TIME = Some(Utc::now().to_string());
   println!("{}", START_TIME);
}
```

Pero el código anterior tampoco compila, el compilador nos avisa de que una variable global mutable solo se puede utilizar dentro de un bloque inseguro *unsafe*. Finalmente, si encerramos el código dentro de *main()* en un bloque *unsafe*, sí que compila:

```
use chrono::Utc;

static mut START_TIME: Option<String> = None;

pub fn main() {
   unsafe{
      START_TIME =  Some(Utc::now().to_string());
      println!("{}",  START_TIME.clone().unwrap());
   };
}
```

Ahora, el código compila y se puede ejecutar, pero no parece una forma muy cómoda de utilizar, aunque en algunas ocasiones pudiera ser útil.

Vamos a ver otro problema relacionado con la extensión del dominio de las variables. En un programa como el anterior, no se necesitaría declarar la variable como global, se podría declarar dentro de la función *main()*. Pero suponga que lo que se quiere es utilizar la variable en un hilo diferente creado dentro de *main()*

```
use chrono::Utc;

pub fn main() {
   let start_time = Utc::now().to_string();

   let thread_1 = std::thread::spawn(||{
      println!("Started {}, called thread 1 {}", &start_time, Utc::now());
   });

   thread_1.join().unwrap();
}
```

Si tratamos de compilar este código, el compilador nos dirá que el hilo *thread_1* podría vivir más tiempo que la variable *start_time* que se ha creado en el marco de datos en el stack de la función *main()* y no está garantizado que la variable sobreviva lo suficiente. Nosotros, viendo el código, sabemos que al hacer el *join()* antes de salir de *main()* estamos garantizando esa supervivencia, pero el compilador no nos deja compartir con otros hilos variables que no tengan una vida útil *&static*.

Hay un par de soluciones posibles sin utilizar variables globales. La primera sería clonar la variable *start_time* y pasar a la closure la propiedad del valor clonado:

```
pub fn main() {
   let start_time = Utc::now().to_string();
   let cloned_start_time = start_time.clone();
   let thread_1 = std::thread::spawn( move ||{
      println!("Started {}, called thread 1 {}", &cloned_start_time, Utc::
   });
   thread_1.join().unwrap();
}
```

Esta solución puede servir para una variable de cadena de caracteres como la anterior, pero si hubiera que clonar una variable de mayor tamaño podría no ser la solución óptima. En esos casos se podría envolver la variable con un puntero *Arc*:

```
use chrono::Utc;
use std::sync::Arc;

pub fn main() {
   let start_time = Arc::new(Utc::now().to_string());
   let cloned_start_time = Arc::clone(&start_time);
   let thread_1 = std::thread::spawn(move ||{
      println!("Started {}, called thread 1 {}", &cloned_start_time,
   });

   thread_1.join().unwrap();
}
```

Si además se necesitara mutabilidad interior de la variable, se podría envolver en un `Arc<Mutex<String>>`.

18.2.1 Valor conocido en tiempo de compilación

Cuando el valor de la variable global se conoce en tiempo de compilación, hay básicamente dos soluciones:

- **const:** valores constantes que se conocen en tiempo de compilación. No permiten la mutabilidad interior. El compilador resuelve sustituyendo el valor en línea (inline)[1].
- **static:** las variables reciben un espacio de memoria en el segmento de datos. Es posible la mutabilidad interior.

Si se necesita mutabilidad interior, para los tipos primitivos se pueden utilizar valores *atomic* [2] y, para tipos más complejos, se pueden usar *locks* en la forma *read-write lock*, *RwLock* o en la forma *mutual exclusion lock*, *Mutex*.

Si lo que se necesita es calcular el valor en tiempo de ejecución, las soluciones con *const* y *static* no sirven.

[1] Conviene consultar el apartado de la cláusula *const* en la documentación en `https://doc.rust-lang.org/std/keyword.const.html` y del concepto de *expresiones constantes* en `https://doc.rust-lang.org/reference/const_eval.html`

[2] Ver libro en líne de Mara Bos `https://marabos.nl/atomics/`

CAPÍTULO 19

Railway Oriented Programming

El *Railway Oriented Programming* es un término acuñado por Scott Wlaschin que propone un modelo del tratamiento de errores en la programación funcional.

- Railway Oriented Programming: `https://fsharpforfunandprofit.com/rop/`
- Monoids without tears: `https://fsharpforfunandprofit.com/posts/monoids-wi`

19.1 Monoides

Morfismo: en varios campos de las matemáticas, se llaman *morfismos* (u *homomorfismos*) a las aplicaciones entre estructuras matemáticas que preservan la estructura interna. Por ejemplo, en teoría de conjuntos, los morfismos son las aplicaciones entre conjuntos; en álgebra lineal, las transformaciones lineales; y en topología, las funciones continuas. En *teoría de las categorías*, el morfismo tiene una noción más general.

Categoría: una categoría viene dada por dos tipos de datos: una clase de objetos y, para cada par de objetos X e Y, un conjunto de morfismos desde X hasta Y. En el caso de una categoría concreta, X e Y son conjuntos de cierto tipo y un morfismo *f* es una función desde X a Y que satisface alguna condición.

Los morfismos se representan frecuentemente como flechas entre los objetos. Esto origina la notación:

$$f : X \to Y \tag{19.1}$$

CAPÍTULO 20

NaN an Inf

https://stackoverflow.com/questions/14682005/why-does-division-by-zero

20.1 Artículos para recomendar a los alumnos

The Mediocre Programmer's Guide to Rust:

https://www.hezmatt.org/~mpalmer/blog/2024/05/01/the-mediocre-programmers-guide-to-rus
html

REFERENCIAS

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. 1994. ISBN 978-0-201-63361-0.

[2] Charles Scalfani. *Functional Programming Made Easier*. Leanpub, June 2021.

[3] Steve Klabnik, Carol Nichols, and Rust Community. The Rust Programming Language: The book. https://doc.rust-lang.org/book.

[4] Rust Foundation. The Rust Standard Library. https://doc.rust-lang.org/std/.

[5] Santiago Higuera de Frutos. *Programación En Rust*. Garceta Grupo Editorial, 2022. ISBN 978-84-17289-88-1.

[6] Visual Studio Code. https://code.visualstudio.com/.

[7] Community of Rust developers. The Rust community's crate registry. https://crates.io/.

[8] The Cargo Book. https://doc.rust-lang.org/cargo/index.html.

[9] UAX #31: Unicode Identifier and Pattern Syntax. https://www.unicode.org/reports/tr31/tr31-37.html.

[10] Wikipedia. CLU (programming language). *Wikipedia*, January 2024.

[11] Eric Normand. *Grokking Simplicity: Taming complex software with functional thinking*. Manning Publications, July 2021.