Introducción a la programación

Un enfoque práctico usando Pascal y Python

Primera Edición

Fernando Bobillo Ortega

Autopublicado

Introducción a la programación

Un enfoque práctico usando Pascal y Python

Primera Edición

Fernando Bobillo Ortega

Autopublicado

ISBN: 978-84-09-60183-7
Depósito Legal: M-10291-2024

A Celia.
A Pablo.

Agradecimientos

Me resulta necesario agradecer a todos los compañeros del Departamento de Informática e Ingeniería de Sistemas que han preparado materiales para la asignatura de Fundamentos de Informática que yo he reutilizado aquí. En particular, me gustaría destacar a José Manuel Colom, Javier Martínez y Santiago Velilla.

Me gustaría agradecer especialmente a Carlos Bobed su ayuda con algunos aspectos del lenguaje Python cuando empezaba a interesarme por el lenguaje. Gracias a él conocí los *type hints*.

También me siento igualmente en deuda con todos los estudiantes que a lo largo del tiempo me han proporcionado una realimentación acerca de la enseñanza de la programación en Pascal.

Por supuesto, me gustaría recordar a todos los profesores que me enseñaron a programar y a los autores de los textos que empleé durante mi formación como ingeniero en informática.

Prefacio

Si bien existen numerosos textos de calidad que facilitan el aprendizaje del lenguaje de programación Python, el propósito de este documento es proporcionar una introducción a Python desde el punto de vista de la didáctica clásica de la programación estructurada. Por ejemplo, aunque Python permite que una variable tenga un tipo dinámico (esto es, que a lo largo de la ejecución de un programa, la variable reciba valores de distintos tipos, como un valor de tipo entero y otro de tipo cadena de caracteres), recomendaremos que a las variables se les asigne siempre un valor del mismo tipo, y presentaremos los *"type hints"* de Python como un mecanismo para conseguirlo. Pensamos que para la adquisición de una formación sólida en metodología de la programación, relativamente independiente del lenguaje de programación, es importante conocer ese tipo de problemas y soluciones clásicas.

El documento está especialmente concebido para los estudiantes de la asignatura de "Fundamentos de Informática" del Grado en Ingeniería Química de la Universidad de Zaragoza. Esta asignatura se ha impartido históricamente utilizando el lenguaje de programación Pascal, como ejemplo concreto donde aplicar los conceptos teóricos. Sin embargo, el paso del tiempo hace que cada vez más alumnos reclamen la renovación del lenguaje de programación elegido. Ciertamente, parece claro que poder aplicar sus conocimientos a un lenguaje más moderno y más de moda como Python, mejoraría la motivación del alumnado.

Para alumnos que ya conocen Pascal, este texto les puede ayudar a adaptar sus conocimientos a Python, al prestar especial énfasis a las diferencias entre ambos lenguajes. Si algún día la asignatura pasase a impartirse en Python, seguro que los (esperemos que pocos) estudiantes repetidores agradecerían la existencia de este texto, que les evitaría tiempo de estudio y asistencia a clases teóricas. A la inversa, quienes hayan recibido una formación más práctica en Python pueden utilizar este documento para adquirir una formación teórica más sólida. En todo caso, pretendemos que el texto sea sucinto y ameno, pero sin renunciar al rigor.

La mayoría de los textos dedicados a la enseñanza de un lenguaje de programación se centran exclusivamente en las capacidades de un lenguaje, pero no tanto en su relación

con otros lenguajes. En el caso concreto de Python, los textos con mucha frecuencia se ocupan de los múltiples aspectos avanzados que incluye Python pero que pueden exceder los contenidos de un curso de introducción a la programación (por ejemplo, las clases y objetos). Para aspectos avanzados sobre estos lenguajes, referimos al lector a los manuales de referencia sobre Pascal [12] y Python [13].

A lo largo del texto se intercalará código en Pascal (en turquesa) y código en Python (en morado). Además, en algún caso, se incluirá algún código incorrecto (en rojo). En la versión digital, ambos códigos se muestran con diferentes colores. En la versión impresa, dependerá de si imprimimos en blanco y negro o en color.

Debemos advertir también de que se ha optado por proponer muy pocos ejercicios para el lector (de los que no se proporciona la solución), porque hemos preferido introducir ejercicios originales diferentes a los que se pueden encontrar en cualquier otro libro de introducción a la programación.

Por último, pero no por ello de menor importancia, cabe mencionar que existen diferentes versiones de cada uno de estos lenguajes. El lenguaje Pascal tiene varios estándares (ISO 7185:1983 y ISO 7185:1990) y varios dialectos (Turbo Pascal, Delphi ...), pero en este texto, consideraremos Free Pascal. Python también tiene varias versiones, pertenecientes a los dos grandes familias Python 2 ó Python 3. En este texto, consideraremos Python 3. La versión 3.7 debe ser suficiente para probar el código de este texto, recordando una vez más que no se usarán todas las funcionalidades de esta versión.

Índice

Capítulo 1

Introducción

1.1. Introducción a la informática

Informática y ordenadores. La palabra informática procede de la unión de "INFORMación" y de "automÁTICA". Puede definirse como la ciencia que estudia el tratamiento automático de la información.

Información se entiende en nuestro contexto de un modo muy amplio: datos numéricos, alfabéticos, multimedia, etc. que sean relevante para resolver problemas de tratamiento automático de la información. Se representarán mediantes símbolos que sean manipulables por un ordenador.

Por *tratamiento de la información* se entiende el procesamiento de unos datos iniciales (de entrada) para obtener unos datos finales (de salida) que son consecuencia lógica de las operaciones realizadas sobre los datos de entrada. En principio, si se repitieran las mismas operaciones sobre los mismos de entrada, se obtendrían los mismos datos de salida. Son una excepción los programas basados en números pseudoaleatorios, que sí que pueden mostrar diferentes comportamientos en diferentes ejecuciones.

Un *ordenador* (o computadora) es una máquina electrónica y programable que se utiliza como herramienta para resolver de forma automática problemas de tratamiento de información. La definición exige que se base en tecnología electrónica, lo que lo diferencias de sus antecesores (máquinas electromecánicas). Además, es importante que sea de propósito general y, por tanto, programable: su funcionamiento se especifica a través de programas. Aunque a veces parece que un ordenador "hace lo que quieren", es importante destacar que en realidad no determina su propio comportamiento, sino que hace lo que alguien le ha ordenado a través de un programa.

En islandés, la palabra para referirse a los ordenadores, *"tölva"* significa "profeta de los números". Esta curiosidad refleja bien el hecho de que es bastante más que una calculadora. Mientras que una calculadora clásica únicamente resuelve operaciones aritméticas, un ordenador es capaz de ejecutar operaciones aritméticas y lógicas (como,

por ejemplo, comparar o copiar símbolos). Las calculadores clásicas tampoco relacionan las salidas de unas operaciones como entradas de las siguientes. Las calculadores programables, en cambio, sí que se asemejan conceptualmente a los ordenadores.

Se considera que el primer ordenador fue el ENIAC (1946), aunque desde mucho antes se habían construido máquinas para resolver problemas aritméticos, dispositivos que no eran electrónicos, dispositivos que no eran propósito general, etc. Alan Turing se considera el padre (¿o el abuelo?) de la informática: en los años 30, antes por tanto de que existieran ordenadores como tales, estudió la computabilidad de los problemas (problemas que se pueden y que no se pueden resolver) y propuso una máquina abstracta, la máquina de Turing, como un equivalente teórico a cualquier ordenador físico que pueda existir.

Estructura y funcionamiento de un ordenador. Cuando se habla de ordenadores, se debe diferenciar entre dos niveles diferentes. El *hardware* es la parte física, como por ejemplo el teclado o la pantalla. En cambio, el *software* es la parte lógica, es decir, el conjunto de programas que se pueden ejecutar en un ordenador. Decía una chiste que el hardware es todo aquello que puedes golpear cuando te enfadas porque el ordenador no se comporta como es debido.

La idea básica del funcionamiento de un ordenador es bastante sencilla. La estructura de los ordenadores se basa en los siguientes componentes:

- una *memoria principal*, que almacena los datos y las instrucciones,
- una *unidad central de procesamiento* (o CPU), que ejecuta instrucciones de programas,
- dispositivos *periféricos* complementarios, incluyendo un sistema de entrada/salida que permita la comunicación con el usuario y
- un sistema de *comunicación* entre los componentes anteriores, a través de *buses*.

La memoria principal está formada por muchísimas celdas o celdillas de memoria, cada una de las cuales tiene un dispositivo electrónico con 2 posibles estados. Esos dos posibles estados se corresponden con una magnitud física, pero nos abstraemos de ello y pensamos en su estado en términos de "encendido" o "apagado" o, directamente, 0 ó 1. Por ello, la unidad mínima de información, es un valor binario, y se denomina *bit* (precisamente como una contracción de *BInary digiT*). Esto implica el uso de código binario para representar internamente la información en los ordenadores y su naturaleza digital (es decir, se representarán valores discretos y no continuos).

Con n bits, se pueden representar 2^n valores diferentes.

Ejemplo 1.

- *Con* 1 *bit se representan* 2 *números:* $\{0, 1\}$
- *Con* 2 *bits se representan* 4 *números:* $\{00, 01, 10, 11\}$
- *Con* 3 *bits se representan* 8 *números:* $\{000, 001, 010, 011, 100, 101, 110, 111\}$

Si representamos números naturales en binario, con n bits podemos representar los pertenecientes al intervalo $\mathbb{Z} \cap [0, 2^n - 1]$. Por ejemplo, el Ejemplo 2 ilustra a qué valores en decimal corresponden los valores en binario para el caso de $n = 3$ bits.

Ejemplo 2. *Para* $n = 3$ *bits, la siguiente tabla muestra la equivalencia entre el número en binario y su valor en decimal:*

Binario	*Decimal*
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Dado que el bit es una unidad de información muy pequeña, se suele trabajar con múltiplos de ella. La siguiente tabla muestra las principales unidades de información, sus abreviaturas y su definición. Es importante mencionar que históricamente se han usado tanto potencias de dos como potencias de diez para definir los múltiplos (por ejemplo, a veces 1 Kilobyte se ha entendido como 1000 bytes y otras veces 1024 bytes), pero lo correcto es utilizar las potencias de diez.

Unidad	Equivalencia
bit (b)	unidad mínima de información
Byte (B)	1 B = $8b$
Kilobyte (KB)	1 KB = $10^3 B = 1000B \approx 2^{10} B = 1024B$
Megabyte (MB)	1 MB = $10^6 B \approx 2^{20} B$
Gigabyte (GB)	1 GB = $10^9 B \approx 2^{30} B$
Terabyte (TB)	1 TB = $10^{12} B \approx 2^{40} B$
Petabyte (PB)	1 PB = $10^{15} B \approx 2^{50} B$

¡Atención! 1. *Es importante no confundir la "b" de bits con la "B" de Bytes. Aunque en informática se usan múltiplos del Byte como medida, las empresas de telecomunicaciones típicamente usan como unidad Mbps (megabits por segundo), que es ocho veces mayor que los MBps y mejor de cara al márquetin. No se preocupe si no sabía lo que eran los MBps, le sucede hasta a los presidentes del Gobierno de España.*

La memoria principal es relativamente rápida, pero es volátil (como la memoria RAM) o no modificable (como la memoria ROM). La memoria RAM es la que se usa habitualmente para almacenar información (datos e instrucciones), mientras que la memoria ROM se usa para programas inalterables (como el programa BIOS que controla el arranque de los ordenadores).

En la divertidísima y muy recomendable serie de televisión "The IT crowd" tenían una broma recurrente: cada vez que les llamaban por teléfono, el departamento de informática contestaba con un "Hola, departamento de informática, ¿ha probado a apagarlo y volverlo a encender?". Este típico chiste sobre la informática tiene su fundamento en la volatilidad de la memoria RAM: al apagar y volver a encender se pierde su contenido, así que si el problema se debía a la existencia de ciertos datos o instrucciones en la memoria, podría haber desaparecido. Por supuesto, no es un truco que funcione siempre.

La CPU o procesador se encarga de ejecutar[1] las instrucciones de un programa informática. Actualmente, la mayoría de los procesadores tienen varios núcleos, por lo que realmente incluyen varias CPUs que pueden trabajar en paralelo, pero se suele hablar de "CPU" en singular para abstraernos de ese detalle. Los componentes del procesador incluyen:

- Contador de programa: indica la próxima instrucción a ejecutar,
- Unidad de control: captura la siguiente instrucción de memoria (según lo indicado por el contador de programa), la decodifica para interpretar qué debe hacer con ella y la ejecuta. Dependiendo de la instrucción, puede usar la unidad aritmético-lógica.
- Unidad aritmético-lógica (ALU): componente especializada en realizar operaciones aritméticas y lógicas usando la información almacenado en los registros del procesador,
- Registros: memorias muy rápidas y de muy poco tamaño. Son las que realmente puede manipular la CPU, que debe traerse localmente los datos que necesite desde la memoria principal.

Los dispositivos periféricos incluyen:

- Memoria secundaria o Almacenamiento externo: discos duros, memorias USB, lectores de CD o DVD ... Aunque desde el punto de vista teórica no sería estrictamente necesaria, en la práctica los ordenadores tienen también una *memoria*

[1] Ejecutar un programa suena bélico, pero la alternativa "correr" un programa suena aún peor.

secundaria que supera las limitaciones de la memoria principal: ni es volátil ni es no modificable y tiene mayor capacidad aunque, eso sí, es significativamente menos rápida que la memoria principal.

- Entrada: teclado, ratón, escáner, micrófono ... Todo aquello que permita enviar información desde el usuario al ordenador.
- Salida: monitor, impresora, auriculares, altavoz ... Todo aquello que permita enviar información desde el ordenador al usuario.
- Comunicaciones: tarjetas de red, módem, enrutador ... Todo aquello que permita comunicar un ordenador con otros ordenadores o dispositivos externos.

Cada vez que hacemos doble click para ejecutar un programa en nuestro ordenador, se busca en el dispositivo de almacenamiento (generalmente en el disco duro), se copia en memoria RAM y la CPU ejecuta las instrucciones del programa repitiendo el siguiente proceso mientras queden instrucciones:

- Leer de memoria la instrucción apuntada por el contador de programa.
- Incrementar el contador de programa.
- Decodificar la instrucción leída.
- Hacer que sea ejecutada.

Para ilustrar la necesidad de decodificar instrucciones, veamos un ejemplo:

Ejemplo 3. *En el MIPS R3000 (procesador de la Play Station I), la instrucción* `001000 00001 00010 0000000101011110` *indica que se debe almacenar en el registro r1 la suma del registro r2 y 350.*

Software. Con respecto al software, debemos distinguir entre software existente (creado por otros) y software creado por nosotros mismos. En la medida de lo posible, usaremos software ya existente siempre que sea posible. Sin embargo, si no existe un software adecuado para resolver nuestros problemas o no se adapta del todo a nuestras necesidades, debemos considerar la necesidad de desarrollar nuestro propio software. Poder conseguir tan ardua tarea es precisamente la motivación de este texto.

Entre el software existente, debemos mencionar el *sistema operativo*, facilitando la utilización del ordenador por parte del usuario, al actuar interfaz entre el hardware y el usuario (y el software de aplicación, utilizado directamente por el usuario) y encargarse de tareas como la gestión del sistema de archivos, la memoria, los procesos, la

seguridad, el control de dispositivos periféricos . . . Ejemplos de sistemas operativos son los Windows (como 10 ó 11), macOS, GNU/Linux (como Ubuntu) y, para dispositivos móviles, Android y iOS. Cabe mencionar que los programas que desarrollemos nosotros también pueden interactuar con el sistema operativo (por ejemplo, veremos que esto sucede para hacer que nuestros programas lean ficheros existentes o creen nuevos ficheros).

El software tiene dos partes diferenciadas. El *frontend* es la interfaz que ve el usuario y con la que interactúa. El *backend*, en cambio, es la parte interna encargada del procesamiento de los datos. Durante el aprendizaje de la programación, nos centramos en el backend, por lo que los programas creados serán poco atractivos visualmente. Sin embargo, lo importante al principio es centrarse en los conceptos importantes de programación; una vez dominados, pasar a desarrollar una interfaz gráfica no es tan complicado.

1.2. Introducción a la programación

Algoritmos. Hoy en día estamos relativamente familiarizados con el concepto de algoritmo porque aparece frecuentemente en los medios de comunicación. Hasta no hace mucho tiempo, los periodistas lo confundían con logaritmo. Un *algoritmo* es una secuencia ordenada, finita y bien definida de pasos para resolver un problema concreto (en nuestro caso, nos interesan problemas de tratamiento de la información). El carácter de bien definido significa que las operaciones se deben definir sin ninguna ambigüedad. Un algoritmo puede verse como una serie de instrucciones a realizar sobre unos datos. El nombre procede del matemático persa Al-Juarismi (siglo IX), considerado el padre del álgebra, aunque existen algoritmos mucho más antiguos (por tanto, son muy anteriores a los ordenadores).

El resultado de un algoritmo es independiente de dónde se ejecute: un ordenador obtendría el mismo resultado que un humano que ejecutara el algoritmo usando papel o lápiz (o pizarra y tiza). Por supuesto, el ordenador es preferible por hacerlo mucho más rápidamente, pero el papel y el lápiz siguen siendo muy útiles para depurar programas.

Cuando se introduce el concepto de algoritmo es habitual recurrir al símil de una receta de cocina. Si un algoritmo ejecuta instrucciones sobre unos datos, en una receta de cocina se realizan acciones sobre unos ingredientes. Concretamente, se aplica una secuencia ordenada de pasos para resolver un problema (nuestra hambre). Para cocinar una tortilla de patatas, hay que pelar las patatas, freírlas en abundante aceite de oliva bastante caliente, escurrirlas para eliminar el exceso de aceite, etc. Evidentemente el orden importa (no intente freírlas en su casa sin haberlas pelado antes) y la secuencia

de pasos debe ser finita (para no morir de inanición). El problema es que la secuencia de pasos de una receta de cocina no está bien definida, ya que incluye términos imprecisos como "abundante" o "bastante caliente". Los humanos manejamos bien este tipo de instrucciones, pero los ordenadores no. De hecho, las recetas para robots de cocina son diferentes y se usan instrucciones como "cocinar durante X segundos a velocidad Y".

El algoritmo de Euclides es uno de los más conocidos, y permite calcular el máximo común divisor de dos números enteros a y b. El algoritmo puede describirse así:

1. Sea a el número mayor y sea b el número menor.
2. Si b es igual a 0, entonces el máximo común divisor es a y termina el algoritmo.
3. En caso contrario, volver al paso 2 tomando como a el valor de b y tomando como b el resto de la división de a entre b.

La primera observación es que la descripción del algoritmo no está perfectamente definida: no queda claro si en el paso tercero el nuevo valor de b se calcula a partir del valor anterior de a (en el paso 2) o a partir del nuevo valor de a (recién actualizado en el paso 3). Esto muestra que el lenguaje natural, por su ambigüedad, no es apropiado para especificar algoritmos. En realidad, la interpretación correcta es la primera: el nuevo valor de b se calcula a partir del valor de a en el paso 2.

Ejemplo 4. *Apliquemos el algoritmo de Euclides para calcular el máximo común de divisor de* 12 *y* 8*:*

- *Paso* 1*:* $a = 12,\ b = 8$
- *Paso* 2*: no se aplica*
- *Paso* 3*:* $a = 8,\ b = 4$
- *Paso* 2*: no se aplica*
- *Paso* 3*:* $a = 4,\ b = 0$
- *Paso* 2*: el máximo común divisor es* 4 *y terminamos*

Programas y lenguajes. Un algoritmo es independiente de cómo se represente: puede expresarse en lenguaje natural (desaconsejado), diagramas de flujo, pseudocódigo, lenguajes de programación, ... Un *programa* es una codificación concreta de uno o más algoritmos. Un *lenguaje de programación* es un lenguaje que permite expresar programas, con una sintaxis y semántica adecuados para su uso por el ordenador.

Se considera que la primera programadora (la primera persona que escribió programas) fue Ada Byron, allá por los años 1842–1843: más de un siglo antes de que existiera el primer ordenador y en unos tiempos en los que la tecnología era insuficiente para fabricar un dispositivo con el que probar sus programas. Ada Byron fue hija de Lord Byron y curiosa y desgraciadamente es bastante menos conocida que él (fuera del mundo de la informática) pese a que su contribución a la humanidad fue muy superior.

El pseudocódigo, como su nombre indica, no usa un verdadero lenguaje de programación, pero se parece. Ejemplos de ello serán los Algoritmos 1 y 2. La ventaja de no usar un lenguaje verdadero es que se permite usar licencias que simplifiquen la presentación, como simplificar las instrucciones, combinar instrucciones de diferentes lenguajes, etc. El inconveniente principal es que no puede usarlo un ordenador.

El lenguaje máquina es el único directamente comprensible por el ordenador. Los únicos símbolos que se usan en el lenguaje son el cero y el uno, por lo que las expresiones del lenguaje son secuencias de ceros y unos, como en el Ejemplo 3. Esto hace que para los humanos sea muy difícil escribir programas (pues hay que traducirlos a un código muy diferente del pensamiento humano), entender programas escritos y encontrar errores en ellos. Además, el lenguaje máquina no es universal, sino que depende del hardware concreto del ordenador. Por todo ello, los programadores humanos no lo utilizan.

Los humanos, en cambio, prefieren lenguajes de mayor nivel, más cercanos al lenguaje natural o a la forma de pensar humana. Un primer paso en esta dirección son los lenguajes ensamblador. También dependen del hardware, y son específicos para alguna familia de procesadores. Sin embargo, proporcionan etiquetas simbólicas para referirse a las instrucciones y a los registros del procesador y no es necesario especificar las posiciones de la memoria en binario.

Ejemplo 5. *Un ejemplo de instrucción en ensamblador, para el MIPS R3000, es* `add.s $f2, $f4, $f6` *e indique se deben sumar los valores almacenados en los registros f4 y f6 y almacenar el resultado en el registro f2.*

Los lenguajes ensamblador siguen siendo de demasiado bajo nivel para los humanos: normalmente el programador quiere sumar dos valores pero no necesita preocuparse de qué registros concretos del procesador se usen en el proceso. A cambio, los programas suelen ser más eficientes que los equivalentes programados en lenguajes de alto nivel, por lo que solamente los utilizan programadores avanzados.

La mayoría de los programadores, y desde luego todos los noveles, utilizan *lenguajes de alto nivel*. Son ejemplos de lenguajes de alto nivel Pascal y Python, pero también C, Java o Ada (llamado así en honor a Ada Byron). Los programas son (algo) menos eficientes, pero es mucho más fácil programar, entender programas ya escritos, depurar,

etc. Los programas son independientes de la máquina, si bien será necesario un proceso de traducción a los lenguajes directamente ejecutables en un ordenador. Veamos un ejemplo de instrucción de suma:

Ejemplo 6. *Las instrucciones* `temperaturaK := temperaturaC + 273.15` *(de Pascal) y* `temperaturaK = temperaturaC + 273.15` *(de Python) permiten sumar dos valores. En ambos casos, se suma* 273.15 *a una temperatura en grados Celsius, por lo que se obtiene la temperatura en Kelvin. Obsérvese que el programador no necesita conocer dónde se almacenan ni las temperaturas ni el valor* 273.15*.*

Compiladores e intérpretes. El programador escribe programas en lenguajes de programación de alto nivel, a los que llamaremos *código fuente*. Dado que el único lenguaje que entiende directamente el ordenador es el lenguaje máquina, esos programas se traducirán a un *código ejecutable*, en binario y dependiente del hardware y/o sistema operativo. Cuando ponemos en marcha un programa (un procesador de textos, un navegador Web, o un programa desarrollado por nosotros), lo hacemos a través su ejecutable.

Una primera opción es usar un *compilador*, que es un programa que traduce un fichero escrito en un lenguaje de programación produciendo un fichero escrito en otro lenguaje. Generalmente, traduce de un lenguaje de alto nivel a un lenguaje máquina ejecutable. Una vez que el compilador ha creado el fichero ejecutable, por supuesto, se puede ejecutar. En realidad la palabra compilador no es probablemente una buena traducción del inglés, pero es el término usado habitualmente en español. El lenguaje Pascal requiere el uso de un compilador.

Otra alternativa es usar un *intérprete*, que realiza la traducción en directo durante la ejecución del programa: antes de ejecutar cada instrucción, la traduce del lenguaje fuente al lenguaje máquina. El lenguaje Python es un lenguaje interpretado aunque, como veremos, algunas herramientas de ayuda a la programación realizan una fase previa de revisión del código para encontrar errores.

Mientras el compilador realiza su tarea, pueden suceder varios casos:

- Si el programa no está correctamente escrito (desde el punto de vista de las reglas léxicas y sintácticas del lenguaje), se generarán uno o varios errores y no se creará ningún fichero de salida. El compilador indica qué errores se ha producido y en qué lugares del código fuente, para que el programador pueda solventarlos.

- Si el programa está correctamente escrito, se creará un fichero de salida. Sin embargo, en ocasiones el fichero de salida puede ir acompañado de algunos avisos

(en inglés *warnings*). Mucha gente suele ignorar los avisos (de las cookies de la Web, de las cajetillas de tabaco ...), pero los del compilador deben tomarse muy en serio: aunque no impiden la creación del fichero ejecutable, suelen poner de manifiesto que se ha cometido un error (por ejemplo, calcular un valor pero luego no hacer nada con él).

Ampliación 1. *En el lenguaje Java se usa un esquema mixto. Es un lenguaje interpretado, pero el código fuente se compila previamente a un código intermedio (*bytecode *o código de bytes), que es el que usa el intérprete de Java durante la ejecución de un programa.*

Por tanto, en un programa puede haber dos tipos de errores: los que suceden en tiempo de compilación (detectados por el compilador) y los que suceden en tiempo de ejecución (no detectados por el compilador). Por cierto, un error de software informático también recibe el nombre de *bug* (en inglés, insecto pequeño), debido a una historia real: una polilla ocasionó un cortocircuito al introducirse en un ordenador y el programa no se ejecutaba como debía.

Cualquier lenguaje, sea de programación o no, tiene tres niveles. El nivel léxico especifica el vocabulario, las palabras del lenguaje. El nivel sintáctico especifica las reglas para formar frases combinando palabras. Por último, el nivel semántico determina el significado de las frases. Un compilador puede detectar errores léxicos y sintácticos, pero no puede detectar errores semánticos (que el programa no hace lo que debe). Esto es similar a lo que sucede con la revisión ortográfica y gramatical de un procesador de textos: puede detectar si una palabra no existe en un idioma por no aparecer en el diccionario (error léxico), y puede detectar si una frase es gramaticalmente incorrecta (error sintáctico), pero no puede detectar si lo que decimos es verdad o si tiene sentido (por ahora, quizá algún día la Inteligencia Artificial permita hacerlo).

Ejemplo 7. *Si escribimos "En un lugar de Marcha de cuyos nombre no kiero acordarme", un procesador encontraría como error léxico "kiero" y como error sintáctico "cuyos nombre", pero no detectaría "Marcha". Para él, "Marcha" es sintácticamente correcto y concuerda gramaticalmente con el resto de la frase, así que le resulta tan plausible que la acción se desarrolle en tierras manchegas como estando de marcha (lo que justificaría un estado capaz de confundir molinos con gigantes). Este error hubiera podido detectarse si el revisor inteligente hiciera una búsqueda automática en Internet pero, si el autor del texto escribe un texto inédito, no podría detectarse un error similar.*

Resolución de problemas. A la hora de resolver problemas de tratamiento de la información, debemos considerar una serie de fases. Son pasos ordenados y finitos, pero

demasiado imprecisos como para considerarse un algoritmo. Además, desde cualquiera de las fases se puede regresar a las fases anteriores para modificar el resultado:

Análisis. Consiste en leer las especificaciones y entender bien el problema a resolver. Por cierto, a las nuevas generaciones, que son mucho mejores que las generaciones anteriores en muchos aspectos (por ejemplo, en dominio de los idiomas o de las tecnologías de la información y la comunicación), les suele costar trabajo la comprensión de textos largos: como cualquier otra habilidad, es algo que necesita práctica.

Un chiste gráfico se preguntaba retóricamente por qué perder algunos minutos resolviendo una tarea cuando puedes perder horas tratando de automatizarla. Cuando estamos aprendiendo a programar, resolver cualquier problema construyendo un programa es un ejercicio positivo e interesante. No obstante, no olvidemos que durante nuestro futuro profesional o personal, antes de construir un programa que resuelva una tarea, debemos valorar si el problema es lo suficientemente complejo como para que el esfuerzo merezca la pena.

Diseño. En esta fase se propondría un algoritmo como solución al problema. Este diseño no tiene por qué realizarse directamente en el ordenador. De hecho, cuando se aprende a programar, se recomienda esbozar las primeras soluciones en papel.

Es muy importante pensar bien la solución antes de empezar a codificarla: ahorra mucho tiempo de desarrollo. Durante mi periodo formativo en Italia, me enseñaron la regla de las 10 P, *"Prima Pensare, Poi Programmare, Perché Programmi Poco Pensate Producono Puttanate"* que podría traducirse como "Primero Pensar, Posteriormente Programar, Porque Programas Poco Pensados Producen Porquería".

Codificación. El algoritmo propuesto se escribe en un lenguaje de programación concreto para obtener el código fuente de nuestro programa. En el ámbito de la informática, a este fase se le denomina también *implementación*.

Compilación. Si el lenguaje de programación lo requiere, compilaremos el programa para obtener el código ejecutable. Si el compilador identifica errores, deberemos volver a la fase (¡o a las fases!) anterior para resolverlos. Y, cada vez que se modifique el código fuente, es necesario volver a compilar. En los lenguajes interpretados, normalmente se puede pasar directamente a la siguiente fase. No obstante, algunos lenguajes interpretados realizan también una fase de compilación. Además, algunos entornos de programación en Python (como Pycharm)

realizan una revisión del código similar a la que efectuaría un compilador, pero sin generar como salida código ejecutable.

Pruebas. Debemos ejecutar el programa para verificar que se comporta como es debido, ya que existen errores que el compilador no es capaz de detectar. Recordemos que un programa hace lo que el programador le ordena, no lo que el programador pretendía ordenarle. Es normal que un programa no funcione a la primera, y eso sucede en las mejores familias (casi todo el software que usamos habitualmente publica regularmente nuevas versiones donde corrigen errores de versiones anteriores).

Es importante que las pruebas abarquen un amplio conjunto de casos, tan grande como sea posible. Suele decirse que incluso un reloj parado da la hora exacta dos veces al día; puede suceder que el programa se comporte bien en algunos casos pero no en otros.

Depuración. Si el programa no se comporta como se debido, debe repararse para que funcione correctamente. Habitualmente, se detectará el funcionamiento incorrecto porque el resultado de una ejecución no es el esperado. Decía un chiste que depurar un software es como resolver un crimen en el que nosotros mismos somos los asesinos.

Para ayudar a comprender qué parte del código produce el problema, es habitual obtener una *traza de ejecución*, que consiste en ejecutar el programa paso a paso, instrucción a instrucción, permitiendo observar poco a poco los cambios que produce la ejecución del programa en los valores de las variables. La traza se puede obtener ejecutando el programa nosotros mismos, con papel y lápiz, o se puede usar un software adecuado.

Mantenimiento. El ciclo de vida del software no termina una vez que se publica una versión estable del código ejecutable, sino que puede haber futuras actualizaciones para mejorar el programa con nueva funcionalidad, corregir nuevos errores que se detecten más tarde, etc. Los usuarios de teléfonos inteligentes ya saben que es rara la aplicación para la que no existe una versión más reciente desde nuestra instalación. De todos modos, a esta fase no le prestaremos más atención en el resto de este texto.

Un *entorno de programación* es un software que facilita el desarrollo de programas integrando un conjunto de herramientas: edición de código fuente, compilación, ejecución, ayuda a la depuración, etc. Por ejemplo, DevPascal (para Pascal) y Pycharm (para Python).

Aunque el código fuente podría editarse con cualquier editor de texto, como el bloc de notas de Windows, es interesante utilizar editores especializados que ofrecen ayuda al programador: usan diferentes colores o tipografías para las diferentes partes del programa, pueden usar autocompletado, etc.

A la hora de evaluar la calidad de los programas desarrollados, podemos tener en cuenta varias características:

Corrección. Hacer lo que se pretende en todos los casos posibles.

Fiabilidad. Ausencia de fallos.

Eficiencia. Consumo de recursos, fundamentalmente tiempo y, en menor medida, memoria.

Claridad. Estar escrito de una forma fácilmente comprensible por un humano.

Simplicidad. Estar codificado de una manera tan simple como sea posible (pero no más).

Modularidad. Estar dividido en módulos o subtareas más sencillas, que se podrían reutilizar en el futuro o sustituir por otros módulos de funcionamiento equivalente.

Generalidad. No resolver únicamente un caso particular, sino también casos más generales.

Portabilidad. Funcionar en varios entornos hardware, sistemas operativos, etc.

1.3. Representación de la información

Ya hemos mencionado la naturaleza digital de los ordenadores y el uso de bits para almacenar información en la memoria. Dado que cada bit tiene dos posibles valores, la información se codificará en un código binario. Vamos a ver en detalle la representación de números enteros (Sección 1.3.1) y de números reales (Sección 1.3.3), pues la representación de información de otro tipo de información también se basa en ellas: por ejemplo, los caracteres se pueden representar como enteros (en los códigos ASCII o Unicode) y los colores como tripletas de números enteros (en el código RGB).

1.3.1. Representación de los números enteros positivos

Antes de abordar la representación de los números enteros en un ordenador, es necesario recordar algunos aspectos de los sistemas de numeración posicionales que aprendimos en el colegio pero que, posiblemente, tenemos un poco olvidados.

En los sistemas de numeración posicionales, cada dígito (o cada uno de los símbolos que pueden usarse para formar un número) tiene un valor que depende tanto del propio dígito como de la posición del dígito.

Ejemplo 8. *En el sorteo diario de la lotería de la Organización Nacional de Ciegos Españoles (ONCE) hay cinco bombos, cada uno de los cuales contiene diez bolas numeradas del cero al nuevo. De cada bombo se extrae una bola para formar un número de cinco dígitos. El primer bombo representa las unidades, el siguiente las decenas, el tercero las centenas, y así sucesivamente. Evidentemente, es importante no solo el valor de cada bola sino también el bombo del cual se extrae.*

Veamos un ejemplo con una base diferente, ¡base cinco!

Ejemplo 9. *El mus es un juego de cartas (de baraja española) donde compiten* 2 *parejas de jugadores. Para anotar la puntuación (hasta* 30–40 *puntos por pareja), se colocan en la mesa un montón de piedras pequeñas (o algún sustituto como monedas, garbanzos, etc.) que van cogiendo los jugadores (o* musolaris*). Si la piedra la tiene un miembro de la pareja, representa un punto, pero si la tiene el otro miembro de la pareja, representa cinco puntos (y entonces se denomina* amarraco*). De este modo, se minimiza el número de piedras que hay que colocar sobre la mesa, dejando más espacio para los cubatas.*[2]

De acuerdo con el teorema fundamental de la numeración, un número natural K se escribe en base b como

$$a_n a_{n-1} \dots a_2 a_1 a_0$$

donde

$$K = \sum_{i=0}^{n} a_i \cdot b^i = a_n \cdot b^n + a_n - 1 \cdot b^{n-1} + \dots + a_2 \cdot b^2 + a_1 \cdot b^1 + a_0 \cdot b^0$$

y, para todo $i \in [0, n]$, $a_i \in \{0, 1, \dots, b-1\}$.

Ejemplo 10. *En base* 10 *(decimal), cada* $a_i \in [0, 9]$*, mientras que en base* 2 *(binario), cada* $a_i \in \{0, 1\}$*.*

Si la base no está clara en el contexto, se escribe como subíndice, es decir, $(a_n a_{n-1} \dots a_2 a_1 a_0)_b$. Esto nos resultará especialmente útil en esta sección, donde manejaremos tanto números en decimal con números en binario.

[2] Para ser precisos, en el mus gana la pareja que consiga más *vacas* (similares a los sets del tenis), cada una de las cuales se compone de varios juegos, cada de los cuales se compone de 30 ó 40 puntos. La puntuación de los puntos se anota con las piedras, con el procedimiento descrito, mientras que la puntuación de las vacas y los juegos se anota mentalmente.

Ejemplo 11. *El número* 12345_{10} *es igual a:*
$1 \cdot 10^4 + 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0 =$
$1 \cdot 10000 + 2 \cdot 1000 + 3 \cdot 100 + 4 \cdot 10 + 5 \cdot 1 =$
$10000 + 2000 + 300 + 40 + 5.$

El dígito 5 *va multiplicada por uno (de ahí el nombre de unidades), el dígito* 4 *va multiplicado por diez (de ahí el nombre de decenas), el dígito* 3 *va multiplicado por cien (las centenas), y así sucesivamente.*

El ejemplo anterior, en base decimal, es bastante intuitivo, pero podemos ver que con otra base la situación es la misma.

Ejemplo 12. *Se venden camisetas frikis con el mensaje "Hay 10 tipos de personas: las que saben binario y las que no". Para los que no hayan entendido el chiste, observemos que* $10_2 = 1 \cdot 2^1 + 0 \cdot 2^0 = 2 + 0 = 2_{10}$.

Ampliación 2. *De acuerdo con lo anterior, en base* 16 *(hexadecimal), cada dígito* a_i *debería estar en el intervalo* $\mathbb{Z} \cap [0, 15]$. *Sin embargo, en la práctica,* $a_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$. *Es decir, los números mayores a* 9 *se sustituyen por letras para evitar ambigüedades:* $A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$ *y* $F = 15$. *Si esto no se hiciera, el número* 110 *podría ser* $1 \cdot 16 + 10$ *ó* $11 \cdot 16 + 0$.

Como vimos en el Capítulo 1, el rango de representación con n bits es de 2^n números, los comprendidos en el intervalo $\mathbb{Z} \cap [0, 2^n - 1]$. Recordemos el Ejemplo 2 que muestra el equivalente decimal de cada número binario para el caso de $n = 3$ bits.

A partir de lo anterior, es fácil diseñar el siguiente algoritmo en pseudocódigo para pasar de decimal a cualquier base b positiva (no solamente a binario). La entrada del algoritmo sería un número K en base decimal y la salida sus dígitos a_i en binario.

$i \leftarrow 0$;
mientras $K \neq 0$ **hacer**
  $a_i \leftarrow$ resto de dividir K entre b;
  $K \leftarrow$ cociente de dividir K entre b;
  $i \leftarrow i + 1$
fin

Algoritmo 1: Transformación de un entero positivo K en base decimal a base b.

Ejemplo 13. *Para el número* $K = 20_{10}$*:*

- *Comenzamos con* $i = 0$
- $a_0 =$ *resto de dividir* 20 *entre* 2 $= 0$

- $K = \textit{cociente de dividir } 20 \textit{ entre } 2 = 10$
- $i = i + 1 = 1$
- $a_1 = \textit{resto de dividir } 10 \textit{ entre } 2 = 0$
- $K = \textit{cociente de dividir } 10 \textit{ entre } 2 = 5$
- $i = i + 1 = 2$
- $a_2 = \textit{resto de dividir } 5 \textit{ entre } 2 = 1$
- $K = \textit{cociente de dividir } 5 \textit{ entre } 2 = 2$
- $i = i + 1 = 3$
- $a_3 = \textit{resto de dividir } 2 \textit{ entre } 2 = 0$
- $K = \textit{cociente de dividir } 2 \textit{ entre } 2 = 1$
- $i = i + 1 = 4$
- $a_3 = \textit{resto de dividir } 1 \textit{ entre } 2 = 1$
- $K = \textit{cociente de dividir } 1 \textit{ entre } 2 = 0$
- *Como* $K = 0$*, terminamos*

Por tanto, el resultado es 10100_2.

Este algoritmo obtiene en cada paso i el dígito a_i, es decir, el $i+1$-ésimo dígito menos significativo[3]. Es muy sencillo de entender aunque tiene un inconveniente evidente: los dígitos se obtienen comenzando por el menos significativo por lo que para ciertas operaciones, como por ejemplo escribir el resultado por pantalla, los dígitos se obtienen en el orden contrario al deseado.

El algoritmo contrario, pasar un número en base b a decimal, consiste en calcular la sumatoria $\sum_{i=0}^{n} a_i \cdot b^i$.

Ejemplo 14. $10100_2 = 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 = 0 + 0 + 4 + 0 + 16 = 20_{10}$.

1.3.2. Representación de números enteros

La representación vista en el apartado anterior no soporta números negativos. Para superar esta limitación, veremos tres representaciones diferentes: signo y magnitud, complemento a uno y complementos a dos.

[3] En lenguaje natural, el dígito menos significativo es el "primer dígito menos significativo", aunque en nuestra notación se denota como a_0.

Signo y magnitud. La representación en signo y magnitud permite representar un número signado de n bits usando:

- 1 bit (el más significativo) para representar el signo: 0 para signo positivo y 1 para signo negativo.
- $n-1$ bits para el valor absoluto (magnitud) del número

Ejemplo 15. *Para $n = 3$ bits, la siguiente tabla muestra la equivalencia entre el número en binario y su valor en decimal:*

Binario	*Decimal*
000	$+0$
001	$+1$
010	$+2$
011	$+3$
100	-0
101	-1
110	-2
111	-3

Un aspecto importante es que hasta ahora no habíamos prefijado el número de bits: aplicábamos el Algoritmo 1 y usábamos tantos bits como dígitos fueran necesarios. En la práctica, esto no es así, porque los ordenadores tienen una capacidad finita y dedican un número de bits fijo para representar internamente la información. Por ello, a partir de ahora, el número de bits se fijará de antemano. Para calcular el valor de la magnitud, se partirá del resultado del Algoritmo 1 y se completará con tantos ceros a la izquierda como sea necesario. Al fijar el número de bits, todos los números tendrán el mismo número de dígitos.

Esta representación es muy sencilla de entender, pero tiene dos desventajas:

- Tiene dos representaciones diferentes para el número 0, que se representa como $+0$ y como -0. Además de estar perdiendo la posibilidad de representar un número diferente, esto hace que, por ejemplo, para comparar si dos variables x e y son iguales, haya que considerar cuatro casos:
  - $x = +0$ e $y = +0$,
  - $x = +0$ e $y = -0$,
  - $x = -0$ e $y = +0$ y
  - $x = -0$ e $y = -0$.

- Además, no permite operar aritméticamente, como veremos en el Ejemplo 16. Es decir, la suma de dos números representados en signo y magnitud no es igual al resultado de expresar en signo y magnitud la suma de los dos números en base decimal.

El rango de representación con n bits es de $2^n - 1$ números, los números comprendidos en el intervalo $\mathbb{Z} \cap [-2^{n-1}+1, 2^{n-1}-1]$. Es decir, a las 2^n permutaciones posibles restamos una porque el cero tiene dos representaciones diferentes.

Ejemplo 16. *Se desea calcular* $20_{10} + (-97_{10})$. *Evidentemente, el resultado esperado es* -77_{10}. *Vamos a pasar* 20 *y* -97 *a binario (suponemos* $1B$ *para representar los números), calcularemos la suma y pasaremos a decimal el resultado obtenido en binario.*

- *Hemos visto en el Ejemplo 14 que* $20_{10} = 10100_2$. *Si usamos* $1B$ *para representar el número, el dígito más significativo sería* 0, *por ser un signo positivo, y para la magnitud añadimos* 2 *ceros por la izquierda al resultado del Ejemplo 14 para que ocupa* 7 *bits. Por tanto, en signo y magnitud,* $20_{10} = 00010100_2$.

- *Podemos comprobar que en signo y magnitud* $-97_{10} = 11100001_2$ *(el primer dígito indica que es un número negativo y la magnitud,* 1100001, *se obtiene con el Algoritmo 1.*

- *El siguiente paso es calcular la suma* $20_{10} + (-97_{10})$[4]*:*

$$\begin{array}{rr} & 00010100 \\ + & 11100001 \\ \hline & 11110101 \end{array}$$

- *Finalmente, vamos a convertir a decimal el resultado obtenido,* 11110101_2. *Como estamos trabajando en signo y magnitud, el dígito más significativo (*1*) indica que es un número negativo. El siguiente paso es transformar la magnitud a decimal:*

$$1110101_2 = 1{\cdot}2^0 + 0{\cdot}2^1 + 1{\cdot}2^2 + 0{\cdot}2^3 + 1{\cdot}2^4 + = 1{\cdot}2^5 + 1{\cdot}2^6 = 1+4+16+32+64 = 117\,.$$

Por tanto, el resultado obtenido en la suma es -117_{10}.

Como hemos obtenido $20_{10} + (-97_{10}) = -117_{10} \neq -77_{10}$, *la representación en signo y magnitud no permite operar aritméticamente.*

¿Qué sucede si intentamos representar un valor que excede nuestra capacidad de representación? Si el resultado de una operación aritmética es demasiado grande o demasiado pequeña, se dice que hay un *desbordamiento*.

[4]La suma en binario es similar a la suma en decimal: $0+0=0$, $0+1=1$, $1+0=1$ y $1+1=0$ y me llevo 1 como acarreo.

Ejemplo 17. *Supongamos que tenemos* 3 *bits e intentamos representar el resultado de sumar* $+3$ *y* $+1$*. Como* $+4$ *es mayor que el máximo número representable (*$+3_{10}$*), si tenemos que limitarnos a* 3 *bits de representación, es evidente que el resultado no puede ser correcto. Concretamente,*

$$\begin{array}{rr} & 011 \\ + & 001 \\ \hline & 100 \end{array}$$

por lo que el resultado de sumar $+3_{10}$ *y* $+1_{10}$ *es* -0_{10} *(*100 *es signo y magnitud), que es un resultado incorrecto.*

Representación en complemento a uno. En esta representación:

- Los números positivos se representan como los números naturales, aplicando el Algoritmo 1 y añadiendo tantos ceros a la izquierda como sea necesario hasta tener n bits.
- Los números negativos se representan con la operación de complemento a la base menos uno: $C1(x) = 2^n - x - 1$, donde n es el número de bits.

Una propiedad interesante del complemento a la base menos uno es que puede calcularse de una manera rápida y sencillo intercambiando cada dígito del número binario por su complementario, es decir, cambiando los unos por ceros y los ceros por unos.

Ejemplo 18. $C1(-97_{10}) = C1(01100001_2) = 10011110$.

Otra propiedad de esta representación es que, al igual que sucedía con los números en signo y magnitud, si un número en complemento a uno empieza por 0 es positivo, y si empieza por 1 negativo.

Como desventaja, se siguen teniendo 2 representaciones para el 0 ($+0$ y -0). Sin embargo, esta representación tiene la ventaja de que sí permite operar aritméticamente.

El rango de representación con n bits es de $2^n - 1$ números, los números comprendidos en el intervalo $\mathbb{Z} \cap [-2^{n-1} + 1, 2^{n-1} - 1]$. Este rango es igual al de la representación en signo y magnitud. Sin embargo, las valores son diferentes. Por ejemplo, el número 10000000 en signo y magnitud corresponde al número -0, mientras que en completo a uno corresponde al número $+127$.

Representación en complemento a dos. En esta representación:

- Los números positivos se representan como los números naturales, aplicando el Algoritmo 1 y añadiendo tantos ceros a la izquierda como sea necesario hasta tener n bits.
- Los números negativos se representan con la operación de complemento a la base menos dos: $C2(x) = 2^n - x$, donde n es el número de bits.

Algunas propiedades interesantes de esta representación son las siguientes:

- Es trivial observar que $C2(x) = C1(x) + 1$. Por lo tanto, para calcular el complemento a dos de un número, podemos intercambiar ceros por unos, y unos por ceros (obteniendo el complemento a uno) y, posteriormente, sumar uno.
- Esta representación también verifica la propiedades de que si un número en complemento a uno empieza por 0 es positivo, y si empieza por 1 negativo.
- Como ventaja frente a las representaciones anteriores, tiene una única representación para el 0: el número tal que todos sus dígitos son 0.
- Además, permite operar aritméticamente, como comprobaremos en el Ejemplo 19.
- El rango de representación con n bits es de 2^n números, los números comprendidos en el intervalo $\mathbb{Z} \cap [-2^{n-1}, 2^{n-1} - 1]$. Este rango es asimétrico en el sentido de que número de valores negativos que se puede representar es diferente (superior en una unidad) al número de valores positivos.

Ejemplo 19. *Calculemos de nuevo* $20_{10} + (-97_{10})$ *pero usando la representación en complemento a dos con* $1B$.

- *Al igual que en el Ejemplo 16,* $C2(20_{10}) = 00010100$
- *Como vimos en el Ejemplo 18,* $C1(-97_{10}) = 10011110$. *Por tanto,* $C2(-97_{10}) = C1(-97_{10}) + 1 = 10011110 + 1 = 10011111$.
- *Para calcular la suma* $20_{10} + (-97_{10})$ *obtenemos:*

$$\begin{array}{rr} & 00010100 \\ + & 10011111 \\ \hline & 10110011 \end{array}$$

- *Por último, convertiremos a decimal el resultado obtenido. Como el resultado obtenido empieza por* 1*, corresponde a un número negativo. Además, como* $10110011 = 10110010 + 1$*, desharemos la operación de complemento a uno a* 10110010*, teniendo que* $10110010 = C1(01001101) = C1(77)$*. Por tanto, el resultado de la suma* $10110011 = C2(-77)$*.*

Es decir, efectivamente $20_{10} + (-97_{10}) = -77_{10}$*.*

Ampliación 3. *Debemos advertir al lector de que no todas las sumas de números en binario se realizan sumando directamente las representaciones en complemento a dos. Por ejemplo, para sumar dos números negativos (cuyo dígito más significativo es el cero), está claro que esta suma "directa" produciría un desbordamiento, pues con los dígitos más significativos tendríamos* $1 + 1 = 0$ *y habría un acarreo igual a* 1*.*

Antes de concluir esta sección, veamos el rango de representación para un caso concreto.

Ejemplo 20. *Para* $n = 3$ *bits, la siguiente tabla muestra la equivalencia entre el número en complemento a dos y su valor en decimal:*

Binario (C2)	*Decimal*
000	+0
001	+1
010	+2
011	+3
100	−4
101	−3
110	−2
111	−1

1.3.3. Representación de números reales en binario

Un real K se escribe en base b como: $a_n a_{n-1} \dots a_2 a_1 a_0 . d_1 d_2 \dots d_m$, donde $K = a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \dots + a_2 \cdot b^2 + a_1 \cdot b^1 + a_0 \cdot b^0 + d_1 \cdot b^{-1} + d_2 \cdot b^{-2} + \dots + d_m \cdot b^{-m}$ y, para todo $i \in [0, n]$ y $j \in [1, m]$, $a_i, d_j \in \{0, 1, \dots, b-1\}$

Ejemplo 21. *El número* 12345.67_{10} *es igual a:*
$1 \cdot 10^4 + 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0 + 6 \cdot 10^{-1} + 7 \cdot 10^{-2} =$
$1 \cdot 10000 + 2 \cdot 1000 + 3 \cdot 100 + 4 \cdot 10 + 5 + 6 \cdot 0.1 + 7 \cdot 0.01 =$
$10000 + 2000 + 300 + 40 + 5 + 0.6 + 0.07.$

El cambio de base se realiza convirtiendo por separado las partes entera y decimal. La conversión a binario de la parte entera ya se describió en el Algoritmo 1. La conversión a binario de la parte decimal se puede realizar mediante el Algoritmo 2:

$i \leftarrow 1$;
mientras $K \neq 0$ **hacer**
  $K \leftarrow K * 2$;
  $d_i \leftarrow parteEntera(K)$;
  $K \leftarrow K - parteEntera(K)$;
  $i \leftarrow i + 1$;
fin

Algoritmo 2: Transformación de la parte decimal K a base b.

Ejemplo 22. $0.6875_{10} = 0.1011_2$. *Mostraremos en color azul cómo se obtienen los dígitos de la parte decimal:*

- *Primera repetición del bucle:* $K = 0.6875 \cdot 2 = 1.3750$ *y* $K = 1.3750 - 1 = 0.3750$
- *Segunda repetición:* $K = 0.3750 \cdot 2 = 0.7500$ *y* $K = 0.75 - 0 = 0.75$
- *Tercera repetición:* $K = 0.7500 \cdot 2 = 1.5000$ *y* $K = 1.5 - 1 = 0.5$
- *Cuarta repetición:* $K = 0.5000 \cdot 2 = 1.0000$ *y* $K = 1 - 1 = 0$

Para pasar a base 10 una parte decimal $(d_1 d_2 \dots d_n)_b$ en base b, se calcula

$$\sum_{i=1}^{n} d_i b^{-i}$$

Ejemplo 23. $0.1011_2 = 0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 1/2 + 1/8 + 1/16 = 0.6875_{10}$.

Veamos seguidamente un caso más complejo que nos ilustrará los problemas de los números reales. Concretamos, veamos qué sucede con el número 0.1.

Ejemplo 24. $0.1_{10} = 0.0\overline{0011}_2$

- $0.1 * 2 = 0.2$
- $0.2 * 2 = 0.4$
- $0.4 * 2 = 0.8$
- $0.8 * 2 = 1.6$
- $0.6 * 2 = 1.2$
- $0.2 * 2 = 0.4$
- $0.4 * 2 = 0.8$
- $0.8 * 2 = 1.6$

- $0.6 * 2 = 1.2$
- $0.2 * 2 = 0.4$
- $0.4 * 2 = 0.8$
- $0.8 * 2 = 1.6$
- $0.6 * 2 = 1.2$
- $0.2 * 2 = 0.4$
- ...

Ahora veamos el camino contrario, truncando a los trece primeros decimales:

Ejemplo 25. $0.0001100110011_2 = 0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} + 0 \cdot 2^{-6} + 0 \cdot 2^{-7} + 1 \cdot 2^{-8} + 1 \cdot 2^{-9} + 0 \cdot 2^{-10} + 0 \cdot 2^{-11} + 1 \cdot 2^{-12} + 1 \cdot 2^{-13} = 0.0999755859375_{10}$.

Como evidentemente $0.0999755859375 \neq 0.1$*, no obtenemos el valor original por lo que ¡hay un error de aproximación al convertir a binario!.*

1.4. Ejercicios

Ejercicio 1. *El código Unicode permite representar los caracteres de casi todos los alfabetos del mundo e incluso emojis. Como veremos en la Sección 4.2, Python lo utiliza. Aunque está en continua expansión para añadir nuevos alfabetos no soportados todavía y los nuevos* emojis *que vayan surgiendo, se estima que actualmente permite soportar unos 1.1 millones de caracteres diferentes. ¿Cuántos bits se necesitarían como mínimo para poder representar todos estos valores?*

Ejercicio 2. *En la excelente novela "Rey blanco" de Juan Gómez-Jurado [3] se puede encontrar este texto:*

> *Porque ésa* (sic) *es la genialidad del programa. Sin el código fuente, aunque cualquier agente enemigo se haga con un terminal que tenga instalado el programa, no le servirá de nada. Sin la conexión al ordenador central, es como tener un ladrillo.*

¿Está de acuerdo con que el ejecutable de un programa es inútil si no se dispone del código fuente?

Ejercicio 3. *En la película "Her" (2013), ganadora de un Óscar al mejor guión original, el protagonista se enamora y tiene una relación con Samantha, un software de Inteligencia Artificial. En la película se refieren a Samantha como un sistema operativo, ¿está de acuerdo con esta denominación?*

Capítulo 2

Primeros programas en Pascal y en Python

Antes de estudiar los conceptos teóricos en detalle, consideraremos un par de ejemplos concretos a modo ilustrativo. Por tanto, en este capítulo veremos nuestros dos primeros programas en Pascal y Python.

2.1. Un primer programa: Hola mundo

La tradición dicta que el primer programa desarrollado cuando se comienza el aprendizaje de un nuevo lenguaje de programación sea el "programa Hola Mundo". Se trata de un programa muy simple que únicamente escriba un mensaje por pantalla que, por costumbre, suele ser el mensaje "¡Hola mundo!". En realidad, es una mala traducción del inglés *"Hello world!"*, pero es de nuevo la costumbre la responsable de que en castellano permanezca su uso y no se actualice a una traducción más correcta como, por ejemplo, "Hola gente".

A pesar de su simplicidad, este programa permite comprobar si tenemos correctamente instalado el entorno necesario para el desarrollo de programas (compiladores, intérpretes, etc.). Además, comparando la forma en que el mismo programa se escribe en diferentes lenguajes, podemos adquirir una aproximación (burda, si se quiere, pero muy rápida) a la complejidad de comenzar a aprender un lenguaje de aproximación.

El programa "Hola Mundo" en Pascal es bastante sencillo si lo comparamos con sus homólogos en los lenguajes C o Java:

```
1 PROGRAM holaMundo;
2 BEGIN
3    write('Hola␣mundo')
4 END.
```

Como vemos, solamente requiere 4 líneas y además son bastante fáciles de entender

si tenemos un nivel básico de inglés. Se define un programa con el nombre `holaMundo`, el programa comienza, se escribe por pantalla el mensaje *"Hola mundo"*, aunque en el programa el texto se delimita por comillas simples (o apóstrofos), y el programa finaliza.

Al conjunto o bloque de instrucciones que se encuentra entre el `BEGIN` y el `END.` se le denomina cuerpo del programa del principal. En este caso, tenemos una única instrucción.

¡Atención! 2. *Observemos que el `END.` del programa principal contiene un punto final y es el único `END` entre los que veremos más adelante que lo lleva.*

Aunque las palabras reservadas del lenguaje (`PROGRAM`, `BEGIN` y `END.`) las hemos escrito en mayúsculas, Pascal no es sensible a las mayúsculas, es decir, sería exactamente igual que se hubieran escrito en minúsculas.

El programa "Hola Mundo" en Python es todavía más sencillo:

```
print("Hola␣mundo")
```

Como vemos, no es necesario dar un nombre al programa ni especificar de manera explícita dónde empieza y dónde termina, ya que se sobreentiende que empieza en la primera línea y acaba en la última.

El programa es tan sencillo que podría pensarse que no hay mucho más que añadir, pero aun así conviene realizar un par de puntualizaciones. Según las recomendaciones de estilo de Python (*"PEP 8 - Style Guide for Python Code"* [14]), todos los programas deben acabar con una línea en blanco, aunque en este texto no lo haremos para ahorrar espacio. Por otro lado, Python sí que es un lenguaje sensible a mayúsculas. Además, en Python, las cadenas de caracteres pueden ir delimitadas por comillas dobles o por comillas simples. Por tanto, otra manera equivalente de escribir el primer programa sería:

```
print('Hola␣mundo')
```

Ampliación 4. *Puede ser frecuente encontrar versiones del programa obsoletas: el código anterior es el correcto a partir de la versión 3 de Python, pero hasta Python 2 la sintaxis de `print` era diferente, sin usar paréntesis. Es decir:*

```
print "Hola␣mundo"
```

2.2. Un segundo programa: transformación de temperaturas

El programa "Hola Mundo" permite confirmar que nuestro entorno está preparado para el desarrollo de programas, pero poco más. Seguidamente consideraremos un programa ligeramente más complejo que, por imitación, nos permitirá desarrollar programas más interesantes. En particular, veremos un programa con más de una instrucción en el cuerpo del programa principal.

El programa en cuestión permite transformar grados Celsius en grados Fahrenheit. La versión en Pascal sería la siguiente:

```
1  { Paso de Celsius a Fahrenheit }
2  PROGRAM CelsiusFahrenheit;
3  VAR
4     Celsius, Fahrenheit: real;
5  BEGIN
6     write('Introducir la cantidad de grados Celsius: ');
7     readln(Celsius);
8     Fahrenheit := (9.0 / 5.0) * Celsius + 32;
9     writeln('Cantidad en Fahrenheit: ', Fahrenheit:1:2);
10    writeln('Pulse Intro para continuar...');
11    readln;
12 END.
```

La línea 1 es un comentario. Los comentarios son texto que obvia el ordenador, por lo que no hacen nada como instrucción, pero resultan muy útil para los humanos, al explicar usando lenguaje natural aspectos sobre lo que hace un programa y cómo lo hace. En Pascal, los comentarios comienzan con el carácter { y acaban con el carácter }, pudiendo abarcar varias líneas.

¡Atención! 3. *Aunque por razones de espacio en este texto los programas van a incluir pocos comentarios, en la práctica es importante utilizarlos profusamente para mejorar la legibilidad del código, haciéndolo más fácilmente comprensible a otros programadores que vayan a leer nuestros códigos (¡entre los cuales suele usar nuestro yo futuro!). En todo caso, el peor comentario no es el inexistente, sino el incorrecto: es importante asegurarnos de que la información que se incluye en un comentario sea veraz para no hacer perder tiempo a quien lea nuestro código.*

En la línea 2 se define el nombre del programa. En las líneas 3–4, se definen las variables que se van a utilizar. En este caso, son 2 variables (`Celsius` y `Fahrenheit`) de tipo real (es decir, sus posibles valores son números reales). Más adelante veremos con detalle qué es exactamente una variable y qué posibles tipos de datos existen en Pascal

y Python; por ahora es intuitivo observar que `Celsius` y `Fahrenheit` nos permitirán manejar cantidades representables mediante números reales. En la línea 5 se indica el inicio del programa, que consta de tres fases: la entrada de datos (líneas 6–7), los cálculos específicos del problema a resolver (línea 8) y la salida de datos (línea 9). Las líneas $10--11$ esperan a que el usuario pulse la tecla Intro para finalizar la ejecución del programa, como indica la línea 12.

A continuación, añadiremos algunas consideraciones sobre este programa:

- El cuerpo del programa principal incluye 6 instrucciones, que se encuentran en las líneas 6, 7, 8, 9, 10 y 11. En el lenguaje Pascal, el carácter `;` permite separar una instrucción de la siguiente. Por ello, la última instrucción del programa principal no tiene por qué acabar con `;` (ya que no tiene una siguiente instrucción de la que haya que separarla). Teóricamente, sería posible que varias instrucciones (por ejemplo, las líneas 6 y 7) estuvieran en la misma línea de un programa, aunque es buena idea evitarlo para mejorar la legibilidad del código. Por otro lado, veremos más adelante algunas instrucciones complejas que ocuparán más de una línea (por ejemplo, condicionales y bucles).

- Las instrucciones del programa principal se ejecutan exactamente en el mismo orden en el que se escriben en el programa. Más adelante veremos que esto no siempre es así, ya que existen mecanismos para complicar el flujo de ejecución de un programa, como los condicionales, los bucles y las funciones.

- Con respecto a la entrada de datos, la línea 7 utiliza la instrucción de entrada[1] `readln` que interrumpe la ejecución del programa, espera a que el usuario escriba un valor usando el teclado, y lo almacena en la variable que se indique como argumento entre paréntesis (en este caso, `Celsius`).

  De este modo, al usuario se le abriría una pantalla para que tecleara un número, pero el usuario tendría que saber que el programa está esperando a que escriba un número para seguir avanzando, de lo contrario, podría pensar que el programa se ha atascado y no funciona bien. Para evitar este tipo de confusiones, antes de utilizar la instrucción `readln`, es buena idea (por decirlo de un modo elegante, pero en la práctica podemos asumirlo como algo obligatorio) escribir un mensaje al usuario informando de lo que se espera que haga, lo que se hace en la línea 6.

- Los cálculos específicos para este problema consisten en aplicar operaciones aritméticas: división (`/`), multiplicación (`*`) y suma (`+`). El resultado se almacena en

[1] Más adelante veremos que es un subprograma, concretamente un procedimiento, predefinido en Pascal.

la variable `Fahrenheit`.

- La salida de datos consiste en escribir por pantalla el valor de la temperatura en Fahrenheit, calculada en el paso anterior, precedida de un mensaje explicativo. La instrucción de salida `writeln` es similar a `write`[2], pero además escribe al final un salto de línea (usaremos una u otra dependiendo del formato deseado para la salida). Ambos subprogramas pueden tener varios argumentos, separados por comas, y todos ellos se escriben por pantalla. Si un argumento está delimitado por comillas simples, quiere decir que se escribirá literalmente esa cadena de texto. Si no va entre comillas simples, se escribirá el valor que tenga esa variable (o esa expresión, en el caso de tener una expresión más compleja).

  En el caso de querer escribir una variable de tipo real, por defecto Pascal utiliza formato científico. Generalmente, es preferible mostrarlo en formato decimal, mucho más sencillo para los alumnos salvo en el caso de números realmente grandes y pequeños. Para escribir una variable `Fahrenheit` con 2 decimales, escribimos `Fahrenheit:1:2`. El último de los números enteros (en este caso, el 2) indica el número de decimales mostrados. El primer número entero indica el número mínimo de caracteres que deseamos que se escriba por pantalla, lo que puede usarse, por ejemplo, para alinear a la derecha datos mostrados en diferentes líneas.

- Si no incluyéramos las líneas 10 y 11 de nuestro programa, una vez escrita por pantalla la temperatura en Fahrenheit, finalizaría la ejecución del programa y se cerraría la ventana de ejecución inmediatamente, sin que al usuario le diera tiempo a ver el resultado. Para evitar esto, la línea 11 hace que el programa permanezca a la espera de que el usuario pulse la tecla Intro (también llamada Return o Enter). Y, al igual que dijimos durante la entrada de datos, para que el usuario sepa qué se espera de él y por qué el programa se ha detenido, previamente la línea 9 escribe por pantalla un mensaje informativo.

Invitamos al lector a que abandone la lectura de este texto (¡temporalmente!) para ejecutar este programa en su ordenador como se describe en el Ejemplo 26.

Ejemplo 26. *Tras ejecutar el programa, el sistema operativo abrirá una ventana en la que se le solicitará que escriba la temperatura en grados Celsius. Supongamos que tiene la fortuna de encontrarse a unos agradables* $20\ ^{o}$ *C. En ese caso, el programa escribirá que la temperatura es de* $(9.0/5.0) * 20 + 32 = 68$ *Fahrenheit. Finalmente, el programa mostrará un mensaje y permanecerá a la espera de que el usuario pulse la tecla Intro,*

[2]Más adelante veremos que ambas son procedimientos, predefinidos en Pascal.

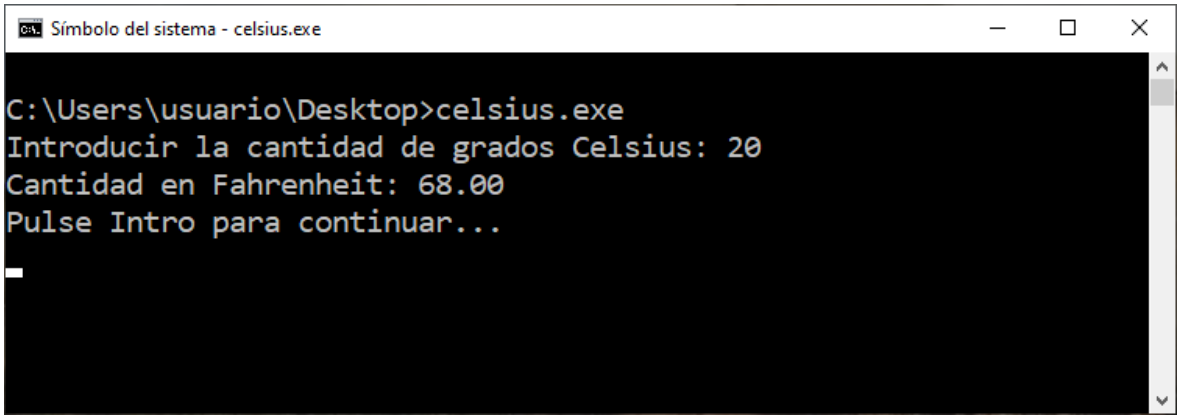


Figura 2.1: Ejecución del programa en Pascal para pasar a grados Fahrenheit.

momento en que finalizará la ejecución del programa. Si el programa se guarda en un fichero llamado celsius.pas*, el compilador producirá un fichero llamado* celsius.exe *que producirá la imagen de la Figura 2.1:*

El programa en Python sería similar, aunque todavía más sencillo, ya que solamente necesita 7 líneas (en realidad 8, si añadimos al final una línea en blanco como recomienda el libro de estilo de Python):

```
1 """ Paso de Celsius a Fahrenheit """
2 Celsius: float
3 Fahrenheit: float
4 print("Introducir␣grados␣Celsius:␣", end='')
5 Celsius = float(input())
6 Fahrenheit = (9.0 / 5.0) * Celsius + 32
7 print("Cantidad␣en␣Fahrenheit:␣", Fahrenheit)
```

En este caso, si el fichero se llama celsius.py, en el entorno Pytorch se obtendría la siguiente salida de la Figura 2.2.

Conviene hacer algunas consideraciones importantes sobre el programa anterior:

- Mientras que el operador de asignación en Pascal es `:=`, en Python es `=`.
- El `writeln` de Pascal es equivalente al `print` de Python; si queremos emular el `write` de Pascal, se debe usar el parámetro `end=''`.
- La lectura de teclado usa un función `input`, que devuelve una cadena de caracteres, y una función `float`, que devuelve a número real la cadena de caracteres leída por `input`. También existe una función `int` como alternativa a `float`, que devuelve un entero.

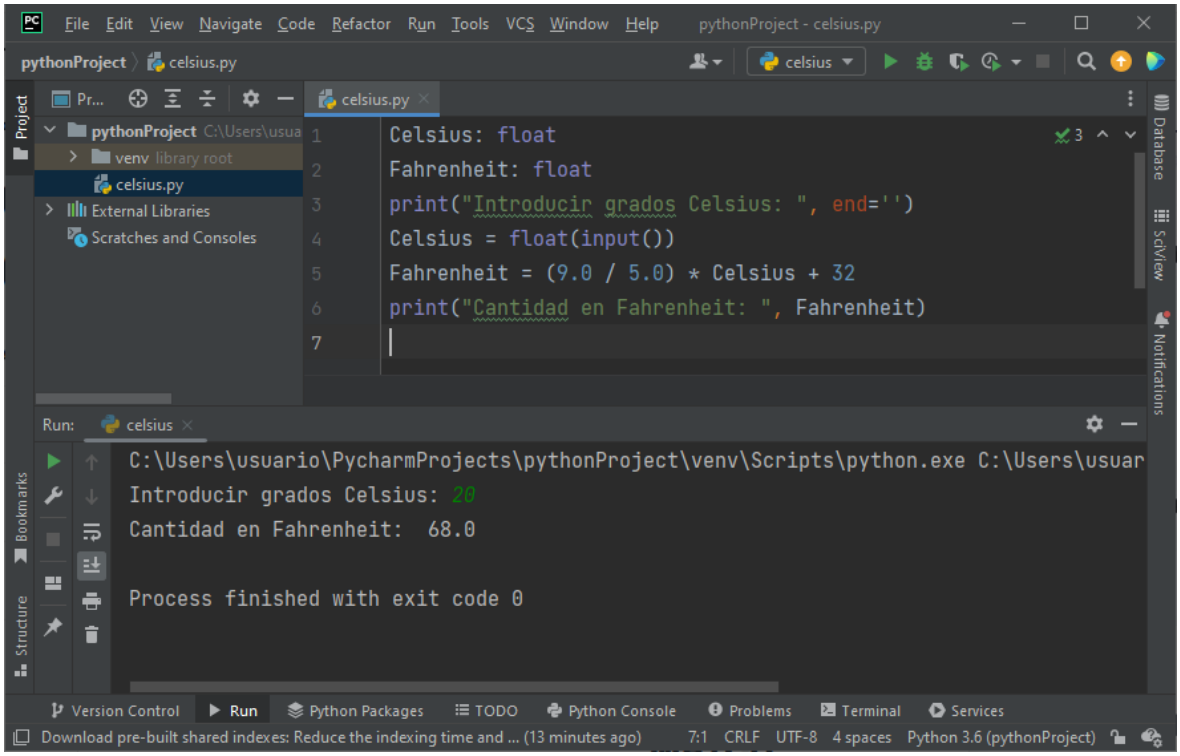


Figura 2.2: Ejecución del programa en Python para pasar a grados Fahrenheit.

- En Python hay dos tipos de comentarios. Por un lado, el carácter # permite definir un comentario que abarca hasta el final de la línea. Por otro, la cadena """ (tres comillas dobles consecutivas) permite definir un comentario que abarca hasta que vuelva a aparecer la misma cadena, pudiendo abarcar varias líneas.

Lo realmente interesante es que podemos hacer muchos programas más manteniendo una estructura similar: definir las variables de entrada, leer su valor de teclado, calcular las variables de salida (posiblemente calculando previamente otros valores intermedios) y mostrar por pantalla el resultado final.

Capítulo 3

Variables y constantes

En un programa se ejecuta instrucciones sobre unos datos. Estos datos se pueden representar como variables (si su valor puede cambiar) o como constantes (si su valor permanece fijo). Veamos ambos casos.

3.1. Variables

3.1.1. Concepto de variable

Nuestro segundo programa en Pascal involucraba el uso de variables y, aunque no las hemos definido formalmente, seguro que el lector no ha echado en falta la definición para entender el funcionamiento del programa. Sin embargo, es conveniente profundizar en el concepto de variable para abordar problemas más complejos.

Recordemos que la memoria de un ordenador almacena datos e instrucciones y que contiene muchísimas celdillas de memoria, cada una con capacidad para representar un bit de información. Hemos visto también que para representar números necesitaremos en general más de un bit. Una manera de bajo nivel de almacenar números en el ordenador sería indicarle qué celdillas de memoria estarían encendidas y cuáles apagadas. Sin embargo, esto tiene varios problemas. Por un lado, necesitaríamos conocer en qué lugar de memoria exactamente se almacenaría la información, lo que no es deseable. Además, las direcciones de las celdillas de memoria se especifican en binario, lo que es poco legible para los humanos y propenso a errores. Por otro lado, necesitaríamos saber cómo se representa la información. Por ejemplo, habría que representar los valores en binario, pero además habría que conocer aspectos como si las celdillas con un menor valor de dirección de memoria representan los dígitos más o menos significativos.

Para evitar todo esto, las variables son una abstracción de varias celdas de memoria a las que se puede acceder con un nombre simbólico y que almacenan un dato cuyo valor se representa con una sintaxis adaptada al programador humano. Por ejemplo, a la variable `Celsius` le podemos asignar el valor `24.0` para reflejar la temperatura de este

hermoso día. El ordenador será quién se encargue de gestionar qué celdas almacenarán en memoria (la *dirección de memoria*) el valor de la variable y la representación interna del valor.

Podemos pensar en una variable como un dato de valor alterable durante la ejecución de un programa con:

- un *nombre simbólico* que permite referenciar la variable;
- un *valor*, alterable pero único en cada momento; y
- opcionalmente, un *tipo de dato* que determina los posibles valores.

Por tanto, hay que distinguir entre el concepto *físico* de variable, que corresponde a un área de una cierta longitud reservada en la memoria principal, y el concepto *lógico* de variable, que corresponde al acceso a un dato en memoria principal a través de un nombre simbólico. En nuestro programas consideraremos este último caso.

Cabe destacar que el concepto de variable es muy diferente al usado en matemáticas, donde se trata de un símbolo que representa un elemento no especificado (tal vez desconocido) de un cierto conjunto.

El nombre de una variable es permanente durante la ejecución de un programa. Algunos lenguajes, como Pascal, exigen que cada variable tenga un tipo de dato invariable. En otros lenguajes, como Python, el tipo de dato sí que puede cambiar (aunque en este texto no usaremos esa posibilidad). El valor de la variable, como ya se ha dicho, puede cambiar durante la ejecución de un programa aunque, en el caso de Pascal o de Python con *"type hints"*, dentro del dominio de valores que determina su tipo.

Observemos que el valor de la variable es único: al cambiar el estado de una celda de memoria para representar un nuevo valor, se pierde el estado anterior. Además, las variables siempre tienen un valor, porque las celdas de memoria estarán siempre encendidas o apagadas. Es muy importante que a todas las variables les asignemos un valor conocido antes de usarlas, porque de lo contrario el funcionamiento de un programa no sería predecible, pues dependería del estado inicial de la memoria. Por ahora, para definir el valor inicial de una variable usaremos los dos mecanismos que hemos visto en nuestro segundo programa: leer de teclado su valor (para `Celsius`) o asignándole el resultado de evaluar una expresión (para `Fahrenheit`).

A algunos profesores de informática les gusta presentar una variable como una "caja" donde guardar cosas (información). Personalmente, no me gusta mucho esta analogía porque no deja claro que en esta caja especial solo se puede guardar una cosa en cada momento (si se guarda un valor, se pierde al anterior), siempre se tiene algo (aunque sea el conjunto vacío), existe una dirección conocida para localizar la caja ...

Me parece algo más acertado pensar en una variable como un conjunto de celdas de una hoja de cálculo: las celdas tienen una dirección de memoria de la forma `B2`, pero el usuario puede definir un nombre simbólico para ellas, y se puede cambiar el valor de una celda pero, al cambiar, se pierde el valor anterior. Además, siempre tienen un valor inicial aunque aparentemente estén "en blanco": por ejemplo, podemos usar una función para contar cuántas celdas en blanco hay en un determinando rango.

3.1.2. Creación de variables

En Pascal, toda variable debe declararse antes de ser utilizada (solamente una vez). La declaración de una variable informa al compilador de cierta información sobre la variable: su nombre y su tipo de dato (en otros lenguajes, también se puede indicar un valor inicial). Para ello, existe una zona de declaración de variables (que comienza con la palabra reservada `VAR`). Toda declaración de variable debe incluir el nombre de la variable y su tipo de dato. Opcionalmente, diferentes variables del mismo tipo pueden declararse juntas en la misma sentencia, aunque también es posible que cada una se defina en su propia sentencia. Por ejemplo, las variables `Celsius` y `Fahrenheit` antes consideradas podrían declararse

```
VAR
    Celsius, Fahrenheit: real;
```

pero también

```
VAR
    Celsius : real;
    Fahrenheit: real;
```

Posteriormente, en la zona de definición de código (delimitada por las palabras reservadas `BEGIN` y `END`), podemos definir el valor inicial de las variables. Por ejemplo:

```
Celsius := 20;
```

Dado que las variables se han declarado de tipo real, el compilador convierte automáticamente estos valores enteros a números reales.

En Python, no es necesario declarar la variable antes de ser utilizada, así que directamente podríamos hacer lo siguiente:

```
Celsius = 20.0
```

Observemos que efectivamente se ha asignado un valor inicial sin haber definido previamente la variable y que no ha sido necesario especificar el tipo de dato, ya que se infiere del valor. En este caso, 20.0 indica que la variable `Celsius` es de tipo real.

Si el valor hubiera sido 20, la variable tendría tipo entero, sin hacerse ningún cambio automático de tipos o valores. Podemos comprobarlo con las siguientes instrucciones:

```
Celsius = 20.0
print(type(Celsius))
Celsius = 20
print(type(Celsius))
```

En Pascal, el tipo de una variable es estático. Es decir, si a lo largo de la ejecución del programa queremos modificar el valor de la variable `Celsius` para darle un valor no numérico, se produce un error de compilación. Por ejemplo, la siguiente instrucción no es válida:

```
Celsius := "a";
```

En Python, el tipo de variable no es estático y puede cambiar sin ningún problema. Por ejemplo, tras la instrucción

```
Celsius = 'a'
```

la variable `Celsius` pasaría a ser de tipo cadena de caracteres (como veremos, en Python no existe el tipo carácter).

La existencia de tipos dinámicos es una característica de los lenguajes de programación que tiene sus ventajas y sus inconvenientes. Sin embargo, pensamos que en un curso de introducción a la programación puede causar más confusiones que beneficios. En esta dirección, en Python se ha añadido una característica opcional llamada los indicadores de tipos o *"type hints"*. Estos indicadores de tipos permiten hacer una declaración de las variables, indicando el nombre y el tipo de dato esperado. Opcionalmente, también se puede dar un valor inicial. Por ejemplo, a continuación se muestra como definir un indicadores de tipos para la variable de tipo real `Celsius`:

```
Celsius: float
```

¡Atención! 4. *Según el libro de estilo del código Python, no debe añadirse espacio en blanco antes del carácter ''':'''.*

De esta manera, algunos compiladores de Python pueden avisar si un programa intenta asignar a una variable un valor incoherente con su indicador de tipo. En este caso, no se produciría un error de compilación y el programa se seguiría pudiendo ejecutar, pero algunas herramientas son capaces de lanzar un aviso al programador. Por ejemplo, si intentamos hacer

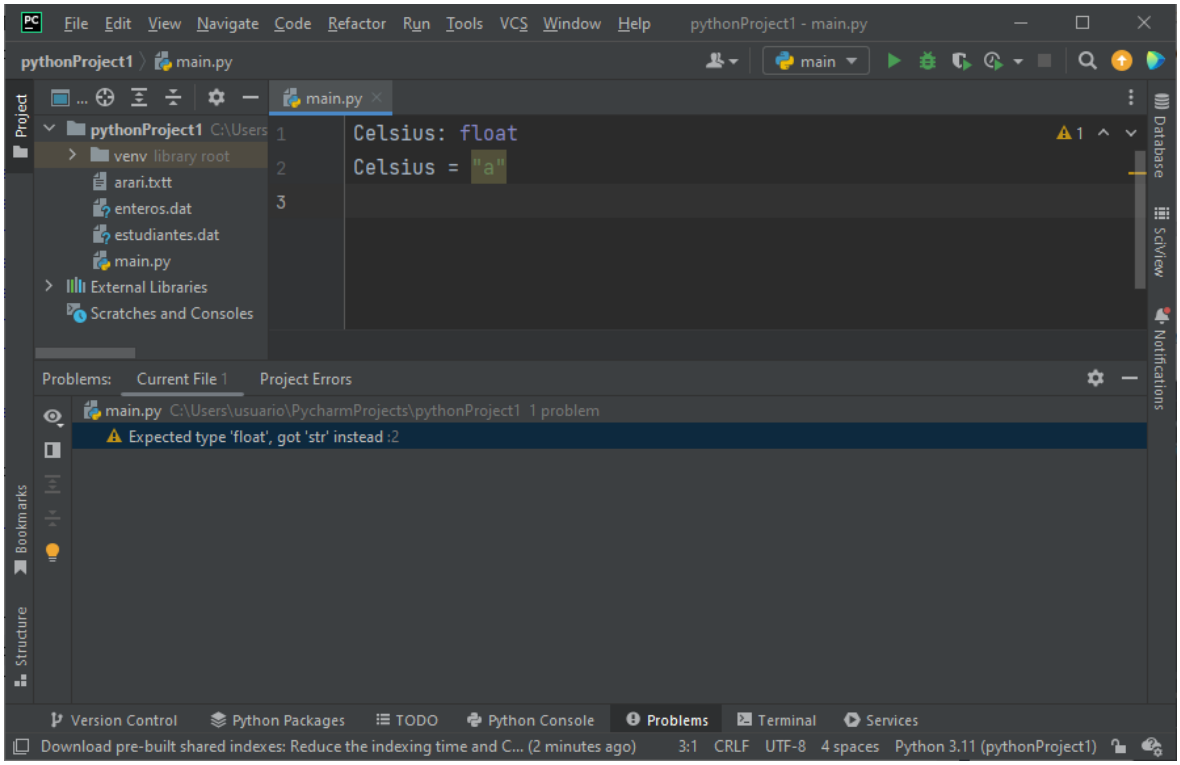


Figura 3.1: PyCharm avisando de una incompatibilidad de tipos.

```
Celsius: float
Celsius = "a"
```

Eclipse PyDev no pondría objeción alguna (obviando completamente el indicador de tipos), mientras que PyCharm sí enviaría un aviso al programador, como se aprecia en la Figura 3.1.

Ampliación 5. *Si queremos trabajar con variables de tipo carácter en Python, los indicadores de tipos nos permiten comprobar que los valores sean cadenas de caracteres, pero no que el número de caracteres sea uno.*

Aunque se usen *type hints*, no es obligatorio que las variables de Python se declaren al principio del programa (como sí sucede en Pascal). No obstante, suele considerarse una buena práctica de programación y nosotros lo haremos.

3.1.3. Nombres de variables

Un *identificador* es un nombre definido por el programador para un elemento de un programa: una variable, una constante, un tipo de dato, un subprograma . . . Aunque al ordenador le da igual el nombre que escojamos, es buena práctica utilizar nombres que faciliten la legibilidad. Por ejemplo, `Celsius` es mucho mejor que `C`.

Declarar una variable es como su "bautizo", al ser el acto donde formalmente se le asigna un nombre. Igual que existen normas sobre los nombres permitidos para bautizar a un hijo (por ejemplo, no se permiten nombres denigrantes), los lenguajes de programación también tienen sus reglas. En Pascal, los identificadores solo pueden contener letras alfabéticos, dígitos numéricos y el guión bajo o subrayado (_) y el primer carácter ha de ser una letra. El caso de Python es similar, pero el primer carácter puede ser una letra o un guión bajo. Por ejemplo, `contador_aminoacidos` o `cantidadPolonio210` serían identificadores válidos.

Igual que un hijo no puede tener el mismo nombre que un hermano suyo que esté vivo, cada nombre debe ser único: una variable no puede tener el mismo nombre que se le haya dado a otro elemento del programa.

Además, existen una serie de palabras que no se pueden usar para los identificadores por ser *palabras reservadas* por el lenguaje para denotar otros elementos del lenguaje. Por ejemplo, `BEGIN` y `END` son palabras reservadas en Pascal.

Por último, por obvio que pueda resultar, nunca está de más insistir en que los nombres de las variables, los programas, los ficheros, etc. hay que tomárselos en serio. No sabemos quién va a acabar leyendo los nombres y, si pensamos cambiarlos en el futuro por nombres más adecuados, puede que se nos olvide hacerlo. Por ejemplo, tuvo cierta repercusión en Internet un auto judicial que mencionaba entre los documentados presentados uno denominado "Demanda de mierda", otro llamado "Informe de mierda número uno" y así un largo etcétera. Además, el funcionario que redactó el auto añadió "(sic)" tras los nombres de los documentos, ¿alguien hubiera podido pensar que los añadió el funcionario de su propia cosecha?

3.1.4. Asignación de variables

En programas anteriores ya hemos usado la asignación para modificar el valor de las variables, pero es el momento de analizarla con algo más de detenimiento.

En Pascal, el operador de asignación es `:=`. En Python, el operador de asignación (`=`) es el mismo operador de comparación de Pascal (`=`), lo que puede causar cierta confusión entre los programadores que conocen ambos lenguajes. En ambos casos, es un operador binario. A la izquierda del operador se coloca una variable y a la derecha una expresión que devuelve un valor del mismo tipo valor que la variable (o compatible con ella).

La semántica del operador es bastante sencilla de contar, aunque a veces al principio es algo costosa de digerir: simplemente se actualiza el valor de la variable que está a la izquierda del operador con el valor de la expresión a la derecha. Recordemos como ejemplo nuestros segundos programas en Pascal y Python donde cambiábamos

la unidad de la temperatura:

```
Fahrenheit := (9.0 / 5.0) * Celsius + 32.0

Fahrenheit = (9.0 / 5.0) * Celsius + 32
```

Consideremos un ejemplo algo más complejo a través del siguiente código en Python:

```
1 a: float
2 b: float
3 b = 1
4 a = 2 * b
5 b = 3
6 print(a)
```

¿Cual sería la salida por pantalla? La línea 4 asigna a la variable `a` el doble del valor de `b`. Sin embargo, en ningún caso "define" que el valor de `a` tenga siempre que ser el valor de `b`. A veces, erróneamente, pensando "a la matemática", algunos programadores noveles asumen que de alguna manera se define una especie de función matemática, de modo que cuando en la línea 5 cambia el valor de `b`, también cambiaría el valor de `a`. Esto no es así en absoluto: en la línea 4 solamente cambia el valor de `a` y en la línea 5 solamente cambia el valor de `b`. Por tanto, por pantalla se escribiría el valor "2".

Veamos otro ejemplo aún más complejo y bastante usado en la práctica: el intercambio del valor de dos variables (que será muy útil, por ejemplo, para ordenar grandes cantidades de números (Sección 9.8). La primera versión que nos puede venir a la cabeza, pero que es incorrecta, es la siguiente:

```
1  PROGRAM intercambiar;
2  VAR
3      a, b : real;
4  BEGIN
5      a := 1;
6      b := 2;
7      a := b;
8      b := a;
9      writeln(a, ' ', b)
10 END.
```

Este programa es incorrecto porque a la variable `a` se le asigna correctamente el valor de `b`, pero cuando a `b` se le asigna el valor de `a` no es el valor que tenía cuando nos interesaba realizar el intercambio, sino después de haberlo actualizado. Por lo tanto, las dos variables acabarían con el mismo valor.

La siguiente versión sí es correcta:

```
PROGRAM intercambiar;
VAR
    a, b, copiaDeA : real;
BEGIN
    a := 1;
    b := 2;
    copiaDeA := a;
    a := b;
    b := copiaDeA;
    writeln(a, '␣', b)
END.
```

La solución ha sido introducir una variable auxiliar, de manera que se guarde el valor inicial de `a` antes de perderlo y que luego se puede asignar a `b` dicho valor.

La traducción directa de este código a Python sería simplemente:

```
a: float
b: float
copiaDeA: float
a = 1
b = 2
copiaDeA = a
a = b
b = copiaDeA
print(a, '␣', b)
```

Sin embargo, Python permite hacerlo de manera simplificada. En el operador de asignación podemos tener a la izquierda y a la derecha varios valores separados por comas, y se garantiza que el resultado sea correcto:

```
a: float
b: float
a = 1
b = 2
a, b = b, a
print(a, '␣', b)
```

3.2. Constantes

Una constante es un dato de valor fijo durante toda la ejecución del programa. De hecho, el valor se puede determinar una única vez y en tiempo de compilación.

Los lenguajes de programación permiten manejar dos tipos de constantes, según se haga un uso directo o un uso simbólico. El uso directo consiste en escribir el valor de la constante en el programa, sin asignarle un nombre previamente. Por ejemplo, las constantes 9.0, 5.0 y 32 en el programa para transformar temperaturas:

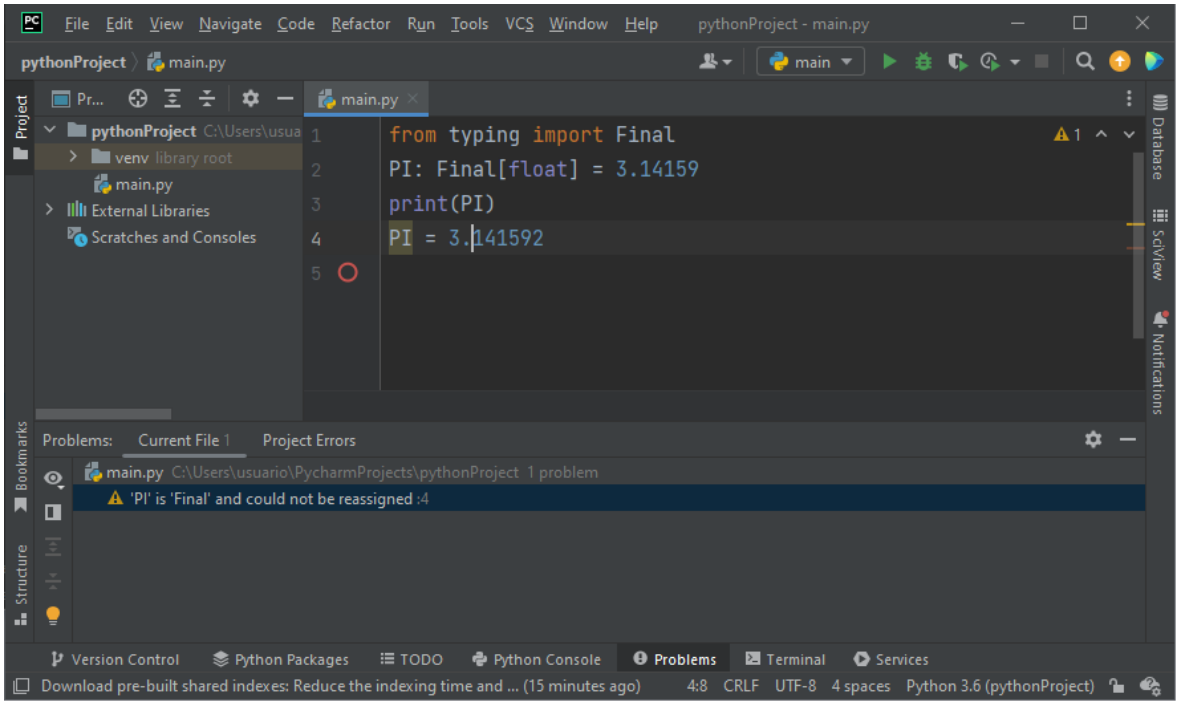


Figura 3.2: PyCharm avisando del cambio de valor de una constante.

```
Fahrenheit := (9.0 / 5.0) * Celsius + 32;
```

El término *literal* hace referencia a estas constantes: es un valor que se escribe directamente en un programa. En el Capítulo 4 veremos la sintaxis de los literales para todos los tipos de datos básicos. A partir del literal, el ordenador puede inferir el tipo de dato (¡como hace Python!). Por ejemplo, "32" es un número entero pero "9.0" es un número real.

Por otro lado, se pueden definir nombres simbólicos para las constantes, y luego se pueden usar esos nombres en el resto del programa. En Pascal, la zona de definición de constantes que comienza por la palabra reservada `CONST`:

```
CONST
   PI = 3.14159;
```

En este caso, se especifican obligatoriamente el nombre y el valor de la constante, pero no el tipo, pues se infiere a partir del valor. Este tipo de constantes es preferible cuando su uso aporta legibilidad a los programas (por ejemplo, una instrucción que calcula el perímetro de un círculo es más fácilmente reconocible si su expresión utilizada la constante `PI`) y permite modificar el valor de la constante que aparece varias veces en un programa en un único lugar (por ejemplo, para añadir más precisión a `PI`, podemos añadir un sexto decimal).

En Python, existen únicamente las constantes simbólicas:

```
Fahrenheit = (9.0 / 5.0) * Celsius + 32
```

Sin embargo, usando la biblioteca `Final`, se pueden definir indicadores de constantes al estilo de los indicadores de variables: si el valor de una constante cambia no se produce un error (ni de compilación ni de ejecución), pero algunos compiladores son capaces de lanzar un aviso al programador.

Por ejemplo, podríamos hacer:

```
from typing import Final
PI: Final[float] = 3.14159
```

El tipo de dato (‘‘`[float]`’’) es optativo, ya que se puede deducir del valor. Por convención, se recomienda que el nombre use letras mayúsculas (y quizá también números y guiones bajos), aunque no es obligatorio.

Al igual que con los indicadores de tipos, si intentamos cambiar el valor de la constante, PyCharm muestra un aviso, pero Eclipse PyDev no. Por ejemplo, la Figura 3.2 muestra un error para el siguiente código:

```
from typing import Final
PI: Final[float] = 3.14159
print(PI)
PI = 3.141592
```

Capítulo 4

Tipos de datos básicos

Un tipo de dato (o, simplemente, tipo) es una agrupación de datos con alguna similitud: valores similares, operaciones definidas similares, etc. Es una idea similar a lo que se hace en matemáticas al definir los conjuntos de los números enteros, reales, complejos, etc.

Los tipos de datos existentes dependen del lenguaje de programación concreto y existen varios criterios de clasificación. En este capítulo, nos centraremos en unos tipos de datos básicos, dejando para más adelante (Capítulo 8) los tipos de datos no predefinidos, definidos por el programador.

Los tipos de datos básicos que consideraremos serán:

- Enteros, como el 32.
- Reales, como el 9.0.
- Caracteres individuales, como 'H'.
- Booleanos o lógicos, con dos posibles valores: verdadero y falso.

Cabe mencionar que, mientras que en matemáticas los conjuntos de los enteros y los reales tienen infinitos elementos, en informática podremos representar solamente un número finito de elementos. Además, en realidad, los lenguajes de programación pueden soportar varios tipos de enteros y varios tipos de reales.

Cada tipo de dato tiene asociadas varias características:

- Un nombre para identificarlo. Por ejemplo, `float` para los reales en Python.
- Un dominio de posibles valores. Por ejemplo, el tipo de dato booleano toma valores en {`True`, `False`}.
- Una representación interna de los valores. Por ejemplo, los enteros se representan en complemento a dos (Sección 1.3.2).

- Una representación algorítmica de los valores a través de literales. Por ejemplo, un carácter en Pascal se representa mediante comillas simples, como 'H'.

- Un conjunto de operadores u operaciones asociados. Por ejemplo, para los tipos de datos numéricos podemos hacer operaciones aritméticas básicas: suma, resta, multiplicación y división.

Se llaman tipos *ordinales* a los que tienen un conjunto finito de valores con una relación de orden definida entre ellos, de manera que es posible contar, comenzando desde un valor inicial, hasta un valor final. En Pascal, los tipos ordinales son los caracteres, los enteros, los tipos enumerados y los subrangos (estos dos últimos de datos se verán en el Capítulo 8).

4.1. Enteros

En Pascal, el principal tipo para manejar enteros es `integer`. El dominio de valores es un subconjunto finito de los números enteros, que depende del número de bytes utilizado. Por lo tanto, podría haber desbordamiento en las operaciones que involucren números de gran magnitud.

Usando n bits, el rango es $[-2^{n-1}, 2^{n-1} - 1]$. Como en Free Pascal el tipo `integer` ocupa $4B$, el rango sería $[-2147483648, 2147483647]$. Para hacer los programas independientes del número de bytes, Pascal define la constante `MAXINT`, cuyo valor es el máximo número entero representable.

¡Atención! 5. *El menor número entero representable en Pascal es* `-MAXINT - 1`.

La representación interna, como ya se mencionó, es en binario en complemento a dos. En cambios, en los algoritmos se utiliza la notación arábiga habitual: una sucesión de dígitos (del 0 al 9), posiblemente comenzando con un `-` que indica un signo negativo.

En Python, en cambio, el tipo de dato entero principal es `int` y permite representar enteros de tamaño arbitrario, por lo que la representación interna es más sofisticada. La representación de los valores también es la notación arábiga usual.

Para los enteros, tenemos definidos los siguientes operadores aritméticos binarios (excepto el cambio de signo que es unario) cuyo resultado es siempre un número entero (excepto en el caso del cociente real, que devuelve un número real, con decimales):

Operador	Pascal	Python
Cambio de signo	`-`	`-`
Suma	`+`	`+`
Resta	`-`	`-`
Producto	`*`	`*`
Cociente entero	`DIV`	`//`
Cociente real	`/`	`/`
Resto o módulo	`MOD`	`%`
Potenciación	✗	`**`

El símbolo ✗ indica que el operador no está definido en ese lenguaje.

También podemos calcular el valor absoluto de un número entero o elevarlo al cuadrado, usando las siguientes funciones:

Operador	Pascal	Python
Valor absoluto	`abs(x)`	`abs(x)`
Cuadrado x	`sqr(x)`	`x ** 2`

Así mismo, disponemos de los siguientes operadores binarios relacionales que devuelven un resultado booleano:

Operador	Pascal	Python
Igual que	`=`	`==`
Distinto que	`<>`	`!=`
Mayor que	`>`	`>`
Mayor o igual que	`>=`	`>=`
Menor que	`<`	`<`
Menor o igual que	`<=`	`<=`

Para los tipos ordinales, Pascal tiene dos funciones para acceder al sucesor y al predecesor:

- `succ`(x) devuelve el elemento siguiente (sucesor) de `x` si existe. La función no está definida para el mayor entero posible (que no tiene sucesor).

- `pred`(x) devuelve el elemento anterior (predecesor) a `x` si existe. La función `pred` no está definida para el elemento menor del conjunto de enteros (que no tiene predecesor).

En Python, no hay un equivalente directo, pero, dado un entero `x`, obviamente, bastaría usar `x + 1` y `x - 1`.

4.2. Caracteres

Este tipo de dato ya no tiene un equivalente directo en las matemáticas, aunque sí en las lenguas habladas por los humanos. En Pascal, los caracteres se representan mediante el tipo de dato `char`. Concatenando varios caracteres, podemos formar una cadena de caracteres, como por ejemplo "Hola mundo". Este tipo de dato lo hemos usado para escribir mensajes por pantalla, pero lo veremos con detalle más adelante (Sección 9.12).

En los lenguajes de programación (y en los ficheros de texto), cada carácter se representa internamente usando un número entero, de acuerdo con una cierta codificación. El dominio de valores que se puede representar viene dado por esa codificación.

En Pascal, los posibles valores son los 128 caracteres de la tabla ASCII, mostrada en la Tabla 4.1. Esta tabla incluye símbolos alfabéticos (como en los lenguajes humanos), no alfabéticos (como signos de puntuación), pero también los dígitos del '0' al '9' y varios caracteres especiales, incluyendo alguno no imprimibles (como un retorno de carro concebido para las antiguas máquinas de escribir). Concretamente, los valores entre 0 y 31 más el 127, representados por acrónimos (excepto el tabulador), son caracteres especiales. Para conocer su significado, puede consultarse [5].

La tabla ASCII está orientada al idioma inglés y ya conocemos a los angloparlantes nativos: no permitieron el manejo directo de elementos del castellano sin un equivalente en inglés como tildes, diéresis, eñes o aperturas de signos de exclamación e interrogación. Pascal usa 1 byte para representar cada carácter, aunque en realidad bastaría utilizar 7 bits para representar esos $2^7 = 128$ valores.

¡Atención! 6. *En Pascal, cuando se escriben mensajes por pantalla, es aceptable omitir deliberadamente las tildes, pues no se soportan en el código ASCII, pero esto no supone una excusa alguna para dejar de escribir las tildes en los comentarios: los comentarios son para los humanos y deben cumplir las normas ortográficas y gramaticales usuales.*

Normalmente, la representación algorítmica es un carácter entre comillas simples, como `'H'`. Sin embargo, para algunos caracteres (como el tabulador), tendremos que hacerlo a través de su valor en la tabla ASCII.

Además del código ASCII, existen otros códigos para representar caracteres. El ASCII extendido (*Extended ASCII*) permite representar 256 valores e incluye caracteres de varias lenguas de Europa occidental, como las tildes del castellano, los acentos graves y circunflejos del francés, las diéresis del alemán, etc. El código Unicode (*Universal code*) dedica más bits (entre 8 y 32) y permite representar todos los alfabetos del mundo

Número	Carácter	Número	Carácter	Número	Carácter	Número	Carácter
0	NUL	32	Espacio	64	@	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	”	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	Tabulador	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Tabla 4.1: Tabla ASCII.

que podamos necesitar (aunque técnicamente no están todos los alfabetos existentes; por ejemplo en 2024 no se soporta el alfabeto dongba para el idioma naxi).

Es buena práctica de programación no suponer en nuestros programas que se usa una codificación concreta de los caracteres como enteros. Los programas serán independientes de la codificación y funcionarían aunque cambiara, y además serán más sencillos de escribir y de leer. La única suposición razonable es que las mayúsculas, las minúsculas y los dígitos numéricos se ubican en posiciones consecutivas. Por ejemplo, en la tabla ASCII las mayúsculas desde las posiciones desde la 65 (‘A’) a la 90 (’Z), las minúsculas van desde la 97 (‘a’) a la 122 (‘z‘) y los dígitos numéricos desde la 48

('0') hasta la 57 ('9').

En Python existen cadenas de caracteres (tipo de dato `str`) pero no el tipo carácter como tal, si bien podría simularse con una cadena de caracteres de tamaño 1. Python usa la codificación Unicode, por lo que podremos escribir mensajes por pantalla empleando tildes sin ningún problema.

Para los caracteres, tenemos los siguientes operadores para pasar de carácter a entero y viceversa:

Operador	Pascal	Python
Paso a char	`chr`	`chr`
Paso a entero	`ord`	`ord`

Por ejemplo, para representar el tabulador en Pascal, podemos usar `chr(9)`.

También tenemos disponibles los operadores relacionales citados para los enteros y, en el caso de Pascal, las funciones `succ`(x) y `pred`(x). En Python, para obtener el sucesor y el predecesor de un carácter `x` se emplearían las siguientes expresiones

```
chr(ord(x) + 1)
```

y

```
chr(ord(x) - 1)
```

respectivamente, recordando que se no se puede calcular el predecesor del primer elemento y el sucesor del último elemento.

4.3. Booleanos

El tipo de dato booleano también es específico de informática. Debe su nombre al matemático George Boole, que propuso una álgebra especial formalizando las operaciones lógicas importantísima en el desarrollo de la informática.

En Pascal, el nombre del tipo de dato es `boolean` y el dominio de valores es el conjunto { `false`, `true` }, que denotan falso y verdadero (respectivamente) y que pueden usarse tal cual en nuestros programas.

Teóricamente, bastaría un bit para representar un valor booleano, pero en la práctica Pascal usa 1 Byte; los ordenadores por eficiencia suelen hacer las operaciones a nivel de bytes.

En Python, el tipo de dato booleano (`bool`) es similar al de Pascal, con valores verdadero (`True`) y falso (`False`). Recordemos que Python es sensible a mayúsculas, mientras que Pascal no lo es.

Veamos ahora las operaciones permitidas, comenzando por los operadores lógicos[1]:

Operador	Pascal	Python
Conjunción	`AND`	`and`
Disyunción	`OR`	`or`
Negación	`NOT`	`not`

La definición de los operadores lógicos se hace a través de unas *tablas de verdad*, que definen el resultado para todos los posibles casos:

a	b	a AND b	a OR b	NOT a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

En realidad, la definición es totalmente intuitiva. Para rellenar la tabla, pensemos, por ejemplo, en cuál sería la respuesta a cada una de las siguientes preguntas:

- ¿Es -1 par Y positivo?
- ¿Es -2 par Y positivo?
- ¿Es 1 par Y positivo?
- ¿Es 2 par Y positivo?
- ¿Es -1 par O positivo?
- ¿Es -2 par O positivo?
- ¿Es 1 par O positivo?
- ¿Es 2 par O positivo?
- ¿Es 2 NO positivo?

En Pascal, para comprobar si una variable está en un determinado intervalo, debemos hacer dos comprobaciones y combinarlas con un `and`:

```
(x >= 0) AND (x <= 10)  { Devuelve true o false }
```

En Python, se puede usar la traducción directa:

```
(x >= 0) and (x <= 10)   { Devuelve True o False }
```

[1]Recordemos que Pascal no es sensible a mayúsculas, por lo que los operadores serían equivalentes en los dos lenguajes

Lo que es más interesante es que Python también permite una sintaxis abreviada:

```
0 <= x <= 10    { Devuelve True o False }
```

Para comprar dos valores booleanos, los únicos operadores relacionales que tienen sentido son la igualdad y la desigualdad:

Operador	Pascal	Python
Igual que	`=`	`==`
Distinto que	`<>`	`!=`

Veremos que los booleanos son muy importantes para formar instrucciones condicionales (Capítulo 5) y bucles (Capítulo 6).

Ejemplo 27. *La expresión booleana* `continuar AND NOT hayError` *es verdadero si y solo si* `continuar` *es verdadero y* `hayError` *es falso. La expresión* `condicion1 OR condicion2` *es cierta si se cumple cualquiera de las dos condiciones. En ambos casos, nos permitirían ejecutar varias veces (mientras las expresiones booleanas sean verdaderas) un conjunto instrucciones.*

¡Atención! 7. *En el lenguaje natural no siempre se utilizan los operadores lógicos de una forma que pueda traducirse directamente a código. Por ejemplo, en un transporte público un cartel rogaba ceder el asiento a "mujeres embarazadas y personas de tercera edad": la expresión booleana "mujeres embarazadas y personas de tercera edad" no va a ser verdadera muchas veces...*

4.4. Reales

Al igual que con el tipo de dato entero, este tipo de dato tiene similitud con el conjunto matemático de los reales, pero también algunas diferencias. Por un lado, el conjunto de los reales en un ordenador no es infinito. Además, como vimos en la Sección 1.3, al manejar números reales en binario, en general, se produce un error de representación.

El nombre del tipo de dato en Pascal es precisamente `real`. El dominio es un subconjunto de los números reales matemáticos pero, a diferencia de los números enteros, no se puede definir un intervalo de modo que se puedan representar exactamente (sin errores de redondeo) todos los valores del intervalo.

La representación interna típica de los números reales se basa en la *coma flotante*. La idea es que la coma se desplaza adaptándose al orden de magnitud del valor que se quiere representar. Por ejemplo, para representar el número 12345.67 se puede desplazar la coma para representarlo como $1234567 * 10^{-4}$, lo que permite representar por separado

el signo, la mantisa (1234567) y el exponente (−4). En un ordenador, la operación usaría una potencia de dos y no una base decimal.

En Python, además del tipo de dato más habitual para representar números reales (`float`), que representa números en coma flotante y, por tanto, con un error en la representación, existe otra alternativa (`Decimal`), que permite representar números reales con un número finito de decimales.

En nuestros programas podemos escribir los valores reales en la notación decimal usual (por ejemplo 12345.67) o en notación científica (por ejemplo, 1234567E–4).

Al trabajar con números enteros debemos tener en cuenta que el coste computacional es superior al de los números enteros y la posible falta de exactitud en los resultados (por ejemplo, al multiplicar un número real puede multiplicarse también su error de representación). A efectos prácticas, tengamos en cuenta que, por ejemplo, puede ser preferible comparar si la diferencia entre dos valores reales es muy pequeña a comparar su igualdad.

Pasemos ahora a citar las operaciones definidas para los números reales. Para empezar, tenemos definidos los mismos operadores aritméticos binarios que para los números enteros. Como excepción, en Pascal no se puede calcular el cociente entero ni el resto de dos números reales, pero en Python sí.

Para los números reales tenemos algunas (en Pascal) o muchas (en Python) funciones definidas en el lenguaje. La siguiente tabla incluye algunos ejemplos importantes:

Operador	Pascal	Python
Valor absoluto	`abs(x)`	`abs(x)`
Cuadrado x	`sqr(x)`	`x ** 2`
Raíz cuadrada	`sqrt(x)`	`math.sqrt(x)`
Seno	`sin(x)`	`math.sin(x)`
Coseno	`cos(x)`	`math.cos(x)`
Tangente	✗	`math.tan(x)`
Arcoseno	✗	`math.asin(x)`
Arcocoseno	✗	`math.acos(x)`
Arcotangente	`arctan(x)`	`math.atan(x)`
Logaritmo	✗	`math.log(x, base)`
Logaritmo neperiano	`ln(x)`	`math.log(x)`
Función exponencial	`exp(x)`	`math.pow(e, x)`
Potenciación	✗	`math.pow(x, y)`
Redondeo por truncado	`trunc(x)`	`math.trunc(x)`
Redondeo al más próximo	`round(x)`	`round(x)`
Redondeo a la parte entera	✗	`math.ceil(x)`
Redondeo hacia abajo	✗	`math.floor(x)`

Cabe destacar que para usar algunas de las funciones anteriores en Python es necesario importar previamente la biblioteca matemática:

```
import math
```

En Python hay dos maneras de calcular la potenciación: con el operador `**` y con la función `math.pow`. Sin embargo, `math.pow` siempre devuelve un número real, mientras que la potenciación con `**` de dos números enteros devuelve otro entero.

Notemos que en Pascal no hay funciones para calcular la tangente, la potenciación arbitraria, el logaritmo de base arbitraria o el redondeo clásico, ya que `round` utiliza el "redondeo del banquero"[2] que redondea hacia el número par cuando el número real es igual de cercano a dos números enteros, de forma que $round(1.5) = 2$ (redondeo hacia arriba) y $round(2.5) = 2$ (redondeo hacia abajo). Veamos algunas soluciones:

- La tangente

  ```
  math.tan(x)
  ```

  de Python podría ser en Pascal, si $x \neq \frac{\pi}{2} + k\pi$ para un número natural k,

  ```
  sin (x) / cos (x)
  ```

- El logaritmo

  ```
  math.log(x, base)
  ```

  de Python podría ser en Pascal, si $x > 0, b > 0$, y $b \neq 1$,

  ```
  ln(x) / ln(base)
  ```

- La potenciación

  ```
  base ** exponente
  ```

  de Python podría ser en Pascal, si la base es mayor que cero,

  ```
  exp (exponente * ln(base))
  ```

- El redondeo clásico

  ```
  round(x)
  ```

  de Python podría ser en Pascal, si la base

  ```
  trunc (x + 0.5)
  ```

El listado completo de funciones matemáticas de Python puede consultarse en [10].

[2] Aunque, evidentemente, este nombre no se corresponde con la realidad de los banqueros, que siempre barren para casa.

4.5. Expresiones

Al igual que en matemáticas, los elementos (constantes, variables, funciones y operadores) pueden combinarse formando *expresiones*, como por ejemplo

```
(9.0 / 5.0) * Celsius + 32
```

En general, todos los elementos de una expresión deben ser del mismo tipo, aunque también se permiten tipos compatibles. Por ejemplo, los enteros se pueden convertir automáticamente a reales, como sucede en el ejemplo anterior con el valor `32`.

Para evaluar las expresiones, existen unas reglas de precedencia (por ejemplo, las multiplicaciones se evalúan antes que las sumas). En concreto, los operadores se agrupan en niveles de prioridad. La siguiente tabla muestra los niveles de prioridad de Python, donde 1 indica máxima prioridad y 10 mínima prioridad:

Nivel prioridad	Operadores	Tipo
1	`(`, `)`	Paréntesis
2	`**`	Exponentes
3	cambio de signo	Signo
4	`*`, `/`, `//`, `%`	Multiplicativos
5	`+`, `-`	Aditivos
6	`==`, `!=`, `>`, `>=`, `<`, `<=`	Relacionales
7	`not`	Negación lógica
8	`and`	Conjunción lógica
9	`or`	Disyunción lógica
10	`=`	Asignación

La exponenciación, la negación lógica y la asignación se evalúan de derecha a izquierda. Los demás operadores binarios, se evalúan de izquierda a derecha.

Se pueden usar paréntesis para modificar el orden de evaluación de las expresiones, al igual que en las expresiones matemáticas. Por ejemplo, `4 ** 3 ** 2` se evalúa (de derecha a izquierda) como $4^{3^2} = 262144$. En cambio, `(4 ** 3) ** 2` se evalúa como 4096. Conviene usar paréntesis solo cuando es estrictamente necesario o cuando mejora la legibilidad de un programa (evitando que el lector tenga que recordar los niveles de prioridad), pero tampoco conviene abusar porque el exceso de paréntesis dificulta la legibilidad.

La siguiente tabla muestra los niveles de prioridad en Pascal. Podemos observar que los operadores lógicos tienen la misma prioridad que ciertos operadores aritméticos: `NOT` tiene la misma prioridad que el cambio de signo, `AND` la misma que los operadores multiplicativos, y `OR` la misma que los aditivos. Además, recordemos que no existe el operador de exponenciación:

Nivel prioridad	Operadores	Tipo
1	(,)	Paréntesis
2	NOT, cambio de signo	Unarios
3	*, /, DIV, MOD, AND	Multiplicativos
4	+, -, OR	Aditivos
5	=, <>, >, >=, <, <=	Relacionales
6	:=	Asignación

4.6. Otros tipos de datos

En Python, existen tipos de dato adicionales, por ejemplo, para representar números complejos (`complex`) y fracciones (`Fraction`).

En Pascal, para representar un número complejo o una fracción utilizando datos básicos, se requiere manejar una pareja de números reales[3]. Por ejemplo, usando dos variables reales `parteEntera` y `parteImaginaria` que podrían usarse para escribirlas por pantalla así:

```
writeln('Complejo: ', parteReal, ' + ',
    abs(parteImaginaria), ' i ')
```

Ampliación 6. *Este código supone que la* `parteImaginaria` *no es negativa, porque en tal caso se imprimirían un signo ‘‘+’’ y luego otro signo ‘‘-’’ antes de la magnitud de la parte imaginaria. Usando la instrucción condicional, que veremos en el Capítulo 5, sería sencillo evitar que si* `parteImaginaria` *es menor que 0 se escribiera la cadena de caracteres ‘‘ + ’’.*

En Python, podemos crear un complejo de la forma

```
complejo = complex(parteReal, parteImaginaria)
```

que puede escribirse por pantalla de la manera usual

```
print("Complejo: ", complejo).
```

De manera similar, mientras que en Pascal necesitaríamos dos variables reales `numerador` y `denominador` para manejar una fracción, en Python podemos crear una fracción:

```
from fractions import Fraction
fraccion = Fraction(numerador, denominador)
```

[3]En el Capítulo 8 veremos que se puede utilizar un tipo de dato estructurado compuesto por dos números reales, como un registro o un array.

Capítulo 5

Condicionales

Según el teorema del programa estructurado (Böhm-Jacopini, 1966), todo programa puede escribirse utilizando las instrucciones de control secuencial, condicional e iterativo. Hasta ahora hemos visto únicamente el control secuencial, pero en este capítulo veremos el condicional y en el Capítulo 6 veremos el iterativo.

La composición condicional se utiliza cuando hay varios casos particulares, cada uno de los cuales requiere unas instrucciones diferentes.

5.1. Única alternativa

La instrucción condicional básica permite ejecutar un conjunto de instrucciones (llamado el *cuerpo* o el *bloque* del condicional) solo si se cumple una condición booleana. Por ejemplo, el programa de transformación de la temperatura a grados Fahrenheit se puede modificar para que solo funcione cuando la temperatura es correcta (mayor e igual que el cero absoluto, 273.15 grados Celsius). En Pascal, la instrucción condicional sería:

```
IF Celsius >= -273.15 THEN
BEGIN
    Fahrenheit := (9.0 / 5.0) * Celsius + 32;
    writeln('Cantidad␣en␣Fahrenheit:␣', Fahrenheit:1:2);
END;
```

Es decir, si `Celsius >= -273.15` es verdadero, se transforma a Fahrenheit y, si es falso, no se hace nada especial. `BEGIN` y `END` no son necesarios si el bloque del `IF` contiene una única instrucción; en ese caso sería preferible, por simplificad, no incluirlos. Es buena práctica de programación *indentar* (o *sangrar*) el bloque del `IF` porque mejora la legibilidad de nuestros programas para los humanos, pero no es obligatorio para el ordenador.

En Python, en cambio, el código anterior sería:

```
if Celsius >= -273.15:
    Fahrenheit = (9.0 / 5.0) * Celsius + 32
    print("Cantidad␣en␣Fahrenheit:␣", Fahrenheit)
```

Un aspecto importante es que en Python no se usan delimitadores como `BEGIN` y `END` para especificar el inicio y el fin de un bucle. En cambio, se basa en el sangrado del cuerpo del bucle, por lo que en este caso sí es obligatorio sangrar bien. Los lenguajes que tienen esta característica se dice que cumplen la regla del fuera de juego (*off-side rule*). Por suerte, es mucho más sencilla de entender y menos polémica que la regla del fuera de juego del fútbol.

Por otro lado, en vez de usar la palabra reservada `THEN` después de la condición booleana, se usa el carácter `:` que, gracias a la regla del fuera de juego, no sería necesario teóricamente, pero los autores del lenguaje Python decidieron que fuera obligatorio para mejorar la legibilidad de los programas.

5.2. Doble alternativa

La instrucción condicional puede ampliarse forma que si la condición booleana es verdadera se ejecute un conjunto de instrucciones y, si es falsa, se ejecute otro conjunto de instrucciones. `IF-THEN-ELSE`, además de un disco de la gran banda de rock *"The Gathering"*, es la instrucción de Pascal que permite llevar a cabo la selección con doble alternativa. Un ejemplo de código en Pascal sería:

```
IF Celsius >= -273.15 THEN
BEGIN
    Fahrenheit := (9.0 / 5.0) * Celsius + 32;
    writeln('Cantidad␣en␣Fahrenheit:␣', Fahrenheit:1:2);
END
ELSE
    writeln('Temperatura␣incorrecta');
```

Ahora, si `Celsius >= -273.15` es verdadero, se transforma a Fahrenheit y, si es falso, se escribe por pantalla un mensaje de error. Antes de `ELSE` no se escribe `;` porque este condicional es una instrucción compuesta (`IF-THEN-ELSE`).

Ampliación 7. *En C o Java sí que se escribe* `;` *antes de* `ELSE`*.*

El mismo código en Python sería:

```
if Celsius >= -273.15:
    Fahrenheit = (9.0 / 5.0) * Celsius + 32
    print("Cantidad␣en␣Fahrenheit:␣", Fahrenheit)
else:
    print("Temperatura␣incorrecta")
```

Python también ofrece la posibilidad de simplificar las instrucciones condicionales anidadas. En ocasiones, hay más de dos casos posibles, por lo que tenemos un `if` con un `else` que a su vez tiene dentro otro `if` (que, a su vez, puede tener un `else` con un `if` dentro, y así tantas veces como se desee). Por ejemplo, para diferenciar si la temperatura de entrada se corresponde con algún valor importante para el agua, podríamos escribir:

```
if Celsius == 0:
    print("Temperatura␣de␣fusión del agua")
else:
    if Celsius == 100:
        print("Temperatura␣de␣ebullición del agua")
    else:
        if Celsius == 374:
            print("Temperatura␣crítica del agua")
        else:
            print("Temperatura␣diferente")
```

La instrucción `elif` permite simplicar tales expresiones. `elif` es equivalente a `else: if` con la gran ventaja adicionales de evitar tener que aumentar el nivel de indentación del código. Por ejemplo, el código anterior quedaría:

```
if Celsius == 0:
    print("Temperatura␣de␣fusión del agua")
elif Celsius == 100:
    print("Temperatura␣de␣ebullición del agua")
elif Celsius == 374:
    print("Temperatura␣crítica del agua")
else:
    print("Temperatura␣diferente")
```

Como aspectos avanzados, deben evitarse las siguientes malas prácticas de programación:

- Tener código redundante tanto en el bloque del `IF` como en el bloque del `ELSE`. En la medida de lo posible, si no es código de ejecución condicional, debería sacarse fuera de la instrucción `IF`.

- Tener código vacío o código absurdo (como asignar a una variable el valor de la propia variable) en el bloque del `IF` o en el bloque del `ELSE`. Esto sucede porque, a veces, los programadores noveles olvidan que el bloque del `ELSE` es optativo. Si el código superfluo está en el bloque del `ELSE`, arreglarlo es tan sencillo como quedarse únicamente con el bloque del `IF`. Si el código superfluo está en el bloque

del `IF`, se puede reescribir la instrucción como un `IF` sin `ELSE` que considere el complementario de la condición booleana. Por ejemplo, el código

```
IF Fahrenheit >= 0 THEN
    Fahrenheit := Fahrenheit
ELSE
    println('Temperatura␣incorrecta');
```

se podría reescribir como:

```
IF Fahrenheit < 0 THEN
    println('Temperatura␣incorrecta');
```

5.3. Operador ternario

Algunos lenguajes como Python, pero no Pascal, permiten usar un operador ternario, que permite escribir un `if-else` de forma abreviada. El operador ternario tiene tres componentes (de ahí su nombre): una condición booleana, el valor cuando la condición es verdadera, y el valor cuando la condición es falsa.

Veamos un ejemplo. Para calcular el máximo entre dos variables de tipo entero `a` y `b`, podríamos escribir lo siguiente:

```
maximo: int
if b > a:
    maximo = b
else:
    maximo = a
```

Usando el operador ternario, en cambio, podríamos escribir:

```
maximo: int
maximo = b if b > a else a
```

Es decir, cuando se cumple `b > a`, el valor de `maximo` pasa a ser `b`; en otro caso, el valor de `maximo` será `a`.

5.4. Múltiple alternativa

Por último, cabe mencionar que Pascal incluye una instrucción condicional múltiple: `CASE` (similar al `switch` de C y Java). Esta sentencia permite considerar más de dos casos de una variable no booleana (de tipo entero, carácter, subrango y enumerado[1]). Por ejemplo, el código anterior sería:

[1]Los tipos subrango y enumerado se verán en el Capítulo 8.

```
CASE Celsius OF
    0 : writeln('Temperatura de fusion del agua');
    100 : writeln('Temperatura de ebullicion del agua');
    374 : writeln('Temperatura critica del agua');
    OTHERWISE : writeln('Temperatura diferente');
END;
```

Para los casos específicos `0`, `100` y `374` se indica qué instrucciones ejecutar (podría haber más de una instrucción sin necesidad de usar `BEGIN` y `END`). `OTHERWISE` indica el código a ejecutar si no se cumple ninguno de los casos anteriores.

¡Atención! 8. *Obsérvese que la instrucción* `CASE` *finaliza con un* `END` *que no concluye ningún* `BEGIN` *previo.*

Recordemos que las tildes se omiten deliberadamente en los programas Pascal, porque los caracteres permitidos se limitan a los valores de la tabla ASCII.

Ampliación 8. *En lenguajes como C o Java, después de ejecutar el código asociado a un caso, por defecto, se pasa a ejecutar el código asociado al siguiente caso. Pascal no funciona así: tras ejecutar el código asociado a su caso, finaliza la ejecución de la instrucción* `CASE`*.*

5.5. Ejercicios

Ejercicio 4. *La empresa Microsoft, a través de su cuenta de la red social (entonces llamada) Twitter @windowsdev publicó un tuit en el incluía código fuente para felicitar el año 2022:*

```
if (DateTime.Now.ToString() == "01/01/2022 00:00:00")
{
    Console.WriteLine("Happy New Year!");
} else
{
    Console.WriteLine("It's still 2021..");
}
```

El tuit usa el lenguaje de programación C#. Pensamos que el lector de este texto es capaz de entenderlo, pero por si acaso confirmemos que `if` *y* `else` *son similares a Python, pero en vez de basarse en el sangrado del cuerpo del* `if` *y del cuerpo del* `else`*, se usan las llaves* { *y* } *como delimitadores. Además,* `DateTime.Now.ToString()` *devuelve una cadena de caracteres con la fecha actual y* `Console.WriteLine` *permite escribir por pantalla.*

El código tenía un problema de tal magnitud que tuvieron que retractarse borrando el tuit. ¿Sería capaz de encontrar el fallo?

```
if (
  (bodyTemperature >= 38 && difficultyBreathing) ||
  (bodyTemperature >= 38 && difficultyBreathing && diabetes) ||
  (bodyTemperature >= 38 && difficultyBreathing && cancer) ||
  (bodyTemperature >= 38 && difficultyBreathing && isPregnant) ||
  (bodyTemperature >= 38 && difficultyBreathing && isOver60yearsOld) ||
  (bodyTemperature >= 38 && difficultyBreathing && hepatic) ||
  (bodyTemperature >= 38 && difficultyBreathing && kidneyDisease) ||
  (bodyTemperature >= 38 && difficultyBreathing && respiratoryDisease) ||
  (bodyTemperature >= 38 && difficultyBreathing && respiratoryDisease) ||
  (bodyTemperature >= 38 && diabetes) ||
  (bodyTemperature >= 38 && cancer) ||
  (bodyTemperature >= 38 && isPregnant) ||
  (bodyTemperature >= 38 && isOver60yearsOld) ||
  (bodyTemperature >= 38 && hepatic) ||
  (bodyTemperature >= 38 && kidneyDisease) ||
  (bodyTemperature >= 38 && respiratoryDisease) ||
  (bodyTemperature >= 38 && respiratoryDisease)
) {
  history.replace(`/diagnostico/${provincia}`)
} else if (bodyTemperature >= 38) {
  history.replace('/cuarentena/')
} else if (bodyTemperature < 38) {
  history.push('/diagnostico_bueno/')
} else {
  history.push('/diagnostico_bueno/')
}
```

Figura 5.1: Fragmento de una página web usada durante la pandemia de COVID-19.

Ejercicio 5. *En los primeros meses de la pandemia mundial por COVID-19, el gobierno de cierto país publicó una página web (que hoy ya no está accesible) con un formulario donde el usuario podía seleccionar una serie de síntomas y de situaciones personales, y a partir de ellos se deba una recomendación.*

- *Los síntomas eran fiebre (`bodyTemperature >= 38`) y dificultad respiratoria (`difficultyBreathing`).*
- *La situación personal incluía embarazo (`isPregnant`), cáncer (`cancer`), diabetes (`diabetes`), edad de riesgo (`isOver60yearsOld`), enfermo hepático (`hepatic`), enfermo renal (`kidneyDisease`) y enfermo respiratorio (`respiratoryDisease`).*
- *Las recomendaciones incluían cuarentena, diagnóstico bueno o diagnóstico dependiente de la provincia.*

La Figura 5.1 muestra un fragmento del código fuente. Aunque está en lenguaje Javascript, pensamos que los lectores podrán entender el código sin mayores problemas. La única aclaración necesaria es que `&&` y `||` son los equivalentes de `and` y `or`, respectivamente. ¿Sería capaz de detectar los múltiples errores conceptuales de este código?

Capítulo 6

Bucles

Las estructuras iterativas o de repetición (conocidas popularmente como los bucles) permiten que una instrucción o un grupo de instrucciones se repitan más de una vez.

En el capítulo anterior hemos visto cómo se puede usar una instrucción condicional para evitar que un programa se ejecute cuando la entrada del usuario es incorrecta. En vez de simplemente mostrar un mensaje de error y finalizar la ejecución, otra alternativa sería solicitar al usuario que escribiera de nuevo los valores de entrada. Sin embargo, es posible que los nuevos valores continuaran siendo incorrectos, por lo que habría que pedirle una vez más que los volviera a escribir. Este proceso debería repetirse un número de veces indefinido a priori, pues desconocemos cuántas veces escribirá el usuario valores de entrada incorrectos. ¡Los usuarios pueden ser muy tozudos (incluso fuera de Aragón)!

En Pascal, tenemos dos bucles indefinidos (`REPEAT` y `WHILE`), que no se sabe cuántas veces van a repetirse, y un bucle definido (`FOR`, con cuenta hacia arriba o cuenta hacia abajo), para casos en que el número de repeticiones es conocido. En Python tenemos el bucle `while` y el bucle `for` (más general que el de Pascal).

6.1. Bucles indefinidos

Comencemos considerando el caso de los bucles indefinidos. Un bucle `WHILE` en Pascal que repita la lectura de datos del usuario sería:

```
WHILE Celsius < -273.15 DO
BEGIN
   write('Introducir la cantidad de grados Celsius: ');
   readln(Celsius);
END;
```

Un bucle `WHILE` se repite mientras una cierta condición booleana sea verdadera (en este caso, `Celsius < -273.15`). El `BEGIN` y el `END` únicamente son necesarios si el

cuerpo del bucle incluye más de una instrucción: si no es así conviene evitarlos por simplicidad. Al igual que con los condicionales, en Pascal, el sangrado del cuerpo del bucle es teóricamente optativo, aunque en la práctica es prácticamente imprescindible por legibilidad del código (si fuera por muchos profesores de introducción a la programación, sería obligatorio).

¡Atención! 9. *No se escribe* `;` *inmediatamente después del* `DO`*, pues en ese caso tendríamos un bucle infinito que repite continuamente la instrucción vacía, y además es un error que pasa desapercibido para el compilador.*

Observemos que, antes del bucle, la variable `Celsius` debería tener un valor inicial. Además, en función de ese valor inicial, este bucle podría no ejecutarse ninguna vez. Para evitarlo, dado que en este caso concreto sí queremos que se ejecuta una vez, podemos usar un bucle `REPEAT` así:

```
REPEAT
   write('Introducir la cantidad de grados Celsius: ');
   readln(Celsius);
UNTIL Celsius >= -273.15;
```

Obsérvese que no es necesario usar `BEGIN` y `END`: `REPEAT` y `UNTIL` ya delimitan perfectamente el inicio y el fin del bucle.

El bucle `WHILE` es un bucle indefinido con control a la entrada: se evalúa la condición de parada antes de entrar al bucle, por lo que el cuerpo del bucle podría no ejecutarse ninguna vez. El bucle `REPEAT`, en cambio, es un bucle indefinido con control a la salida, por lo que se ejecuta siempre al menos una vez. Un chiste gráfico afirmaba que el Correcaminos (que estábase al loro), usa un bucle `WHILE` y, mientras no llegaba al filo del precipicio, seguía corriendo, por lo que nunca se caía. El pobre coyote, en cambio, usaba un bucle `REPEAT`: primero corría y luego comprobaba la condición de parada, así que a veces se despeñaba.

En algunos lenguajes de programación, como Python, no existen bucles indefinidos con control a la salida, por lo que es interesante plantearse cómo simularlos con un bucle `WHILE`. Veamos un código equivalente al bucle anterior:

```
write('Introducir la cantidad de grados Celsius: ');
readln(Celsius);
WHILE (Celsius < -273.15) DO
BEGIN
   write('Introducir la cantidad de grados Celsius: ');
   readln(Celsius);
END;
```

Las diferencias son las siguientes: la condición booleana es justamente la contraria, se ha repetido el cuerpo del bucle fuera de él (para garantizar que se ejecuta al menos una vez) y, como cuestión menor, se han añadido `BEGIN` y `END`.

En Python, el bucle `WHILE` sería:

```
while Celsius < -273.15:
    print("Introducir␣grados␣Celsius:␣", end='')
    Celsius = float(input())
```

Recordemos la regla del fuera de juego mencionada en el Capítulo 5: el inicio y el fin del bucle se indican mediante el sangrado del cuerpo del bucle.

Los bucles infinitos son generalmente indeseados. Para evitarlos, en la condición booleana de un bucle `WHILE` debe aparecer alguna variable que cambie de valor dentro del bucle. Es fácil comprobar que esta condición es necesaria (salvo que usen saltos incondicionales) pero no suficiente.

Ampliación 9. *En realidad, a veces los bucles infinitos se pueden usar en combinación con instrucciones de salto incondicional como `GOTO` o `BREAK`, que no veremos en este texto ya que el paradigma de la programación estructurada desaconseja enérgicamente su uso.*

Muchos problemas requieren leer una secuencia de datos tan larga como se quiera, por lo que se debe usar un bucle indefinido para gestionarla. Como la secuencia podría ser demasiado grande como para almacenarse completamente en memoria, se debe leer un dato, procesarlo[1] y pasar al siguiente, hasta que se cumpla una cierta condición de parada que dependerá del problema. Por ejemplo, la secuencia podría corresponder a una frase que acaba con un punto u otro carácter especial. En Pascal, es interesante conocer la función `eoln`, que devuelve un valor booleano: `true` si se ha llegado a un fin de línea o `false` si hay algún dato diferente al fin de línea. Frecuentemente, se emplean bucles de la forma `WHILE eoln = false DO ...`

6.2. Resolviendo problemas matemáticos

Me gustaría proponer un ejemplo más de bucle indefinido, que tiene una historia detrás. Cuando yo era estudiante de Ingeniería Informática en la Universidad de Granada, uno de mis profesores de matemáticas nos enseñó (entre otras muchas cosas) a resolver ecuaciones en congruencias y nos advirtió de que, aunque las ecuaciones pueden resolverse "por la cuenta de la vieja", ya se encargaría él de poner en el examen

[1] En realidad, a veces se requiere considerar en cada momento más de un dato: por ejemplo, se podrían considerar el último dato leído pero también el penúltimo. Eso sí, nunca se podrían considerar todos los datos en el mismo instante.

ecuaciones lo suficientemente complicadas como para que no nos diera tiempo a resolver todo el examen sin usar los métodos estudiados en clase. El profesor motivaba su asignatura asegurando que las matemáticas eran muy útiles porque nos permitían resolver ese problema de manera mucho más rápida que haciéndolo por la fuerza bruta, lo que es absolutamente cierto. Sin embargo, yo apostillo que la programación permite resolver algunos de esos problemas de una manera más rápida y más fácil. Más rápida porque, como veremos, basta con hacer un bucle de muy pocas líneas y más sencilla porque con el paso del tiempo es más fácil recordar cómo plantear un bucle tan sencillo que recordar la resolución de ecuaciones en congruencias.

El problema que nos planteó en el examen (que, por cierto, no aprobaron demasiados) era el siguiente: calcule el menor número natural tal que $98n + 61$ es divisible entre 1003. Veamos cómo resolverlo en Pascal:

```
PROGRAM congruencia;
VAR
  n : integer;
BEGIN
  n := 0;
  WHILE (((98 * n + 61) MOD 1003) <> 0) DO
    n := n + 1;
  write('El␣menor␣numero␣natural␣es␣', n)
END.
```

El equivalente en Python es más sencillo todavía:

```
n: int
n = 0
while ((98 * n + 61) % 1003) != 0:
    n = n + 1
print("El␣menor␣numero␣natural␣es␣", n)
```

La versión en Python podría ser todavía más sencilla si no usáramos *type hints*, porque se ahorraría la primera línea. Creo que queda probado mi argumento: resulta más conveniente usar programación para calcular este número (por cierto, es el 419; demasiado grande como para obtenerse a ojo).

Otro ejemplo donde la programación demuestra ser más rápida para resolver un problema que las matemáticas es el cálculo de probabilidades. Por ejemplo, calcular la probabilidad de obtener una jugada concreta al repartir a un jugador varias cartas de una baraja de manera analítica puede ser más complicado que realizar un programa que cuente, para todas las posibles combinaciones al repartir una baraja, cuántas de ellas cumplen la condición y dividir entre el número total de posibles combinaciones.

De hecho, en [1] se comete un error al calcular analíticamente una probabilidad[2]. En los Ejercicios 14 y 15 propondremos ejemplos de cálculos de probabilidades.

Otra manera de estimar probabilidades consiste en repetir un experimento (cuyo resultado no es siempre el mismo) varias veces y contar la proporción de veces que se obtiene un resultado sobre el número total de ejecuciones. Para que el resultado del experimento no sea siempre el mismo, se pueden usar números pseudoaleatorios: tanto Pascal como Python ofrecen funciones que permiten obtener números (aparentemente) aleatorios.

En Pascal, cada vez que llamemos a la función `random` sin argumentos, se devolverá un número real en $[0, 1)$. Si a llamamos a `random(x)` se devuelve un número aleatorio entero en $[0, x)$. Previamente, se puede llamar a la función `Randomize` (sin argumentos) para que use un generador de números aleatorios diferente en cada ejecución del programa. Veamos por ejemplo cómo simular el lanzamiento de un dado clásico, de seis caras (lo que se aplicará, por ejemplo, en el Ejercicio 8):

```
PROGRAM dado;
VAR
  n : integer;
BEGIN
  Randomize;
  n := Random(7);
  writeln('El␣dado␣muestra:␣', n)
END.
```

En Python tenemos las funciones `random.random()`, que devuelve un real en $[0, 1)$, y `random.randrange(x, y)`, que devuelve un entero en $[x, y]$. Observemos que es Pascal los posibles números enteros aleatorios no incluyen a y, pero en Python sí. Para poder usar estas funciones de Python, hay que importar el módulo de números aleatorios con `import random`. Por ejemplo, para simular el lanzamiento del dado:

```
import random
n: int
n = random.randrange(1, 6)
print("El␣dado␣muestra:␣", n)
```

En este caso, se crea automáticamente un nuevo generador de números aleatorios en cada ejecución del programa.

[2]En la Sección 2.1.2.3, no se tuvo en cuenta el caso de dobles parejas con 4 cartas iguales que no son reyes ni ases para calcular la probabilidad de tener dobles parejas.

6.3. Bucles definidos

Veamos ahora el caso de los bucles definidos. Un bucle `FOR` en Pascal que escriba los números del 1 al 10, suponiendo una variable entera `i`, sería:

```
FOR i := 1 TO 10 DO
BEGIN
   writeln(i);
END;
```

Es decir, el bucle se repite mientras una *variable de control* (en este caso, `i`) toma valores en el intervalo definido entre un valor mínimo y un valor máximo, contando hacia arriba. Es posible que el bucle no se repita ninguna vez, si el valor máximo es menor que el mínimo. En este ejemplo concreto, como el cuerpo del bucle tiene una única instrucción, el `BEGIN` y el `END` son opcionales (y, por tanto, siempre que sangremos bien el código conviene quitarlos por simplicidad).

En general, es recomendable dar nombres legibles a las variables. Sin embargo, es bastante habitual denominar `i` (de iteración) a la variable de control de un bucle `FOR`.

Veamos cómo simular el bucle anterior con un `WHILE`:

```
i := 1;
WHILE i <= 10 DO
BEGIN
   writeln(i);
   i := i + 1
END;
```

Pensar en un bucle `FOR` como en su bucle `WHILE` equivalente es útil para entender que, si el valor inicial de la variable de control ya es superior al valor final, el bucle no se ejecuta ninguna vez.

En Python, el bucle `for` sería:

```
for i in range(1, 11):
    print(i)
```

La función `range(x, y)` devuelve una lista con los valores enteros en $[x, y)$, es decir, entre x, incluido, e y, no incluido. En este caso, `range(1, 11)` devuelve:

```
[1 2 3 4 5 6 7 8 9 10]
```

El primer argumento es optativo. Si no se indica, sería 0.

La función `range` admite un tercer parámetro que indica el incremento que se suma a cada valor de la lista para obtener el siguiente. Por ejemplo, usando como incremento `-1`, podemos escribir los valores del 10 al 1 así:

```
for i in range(10, 0, -1):
    print(i)
```

Para escribir los valores del 10 al 1 en Pascal, podría usarse una cuenta atrás:

```
FOR i := 10 DOWNTO 1 DO
BEGIN
   writeln(i);
END;
```

Por lo tanto, Pascal solo permite un incremento de 1 (si el `FOR` usa `TO`) o -1 (si el `FOR` usa `DOWNTO`). Python es más flexible que Pascal porque permite más maneras de contar, al permitir incrementos diferentes a ir sumando/restando de uno en uno, e incluso que la variable de control tome un conjunto de valores que no sea una progresión aritmética, como por ejemplo $[7, 4, 8, 3]$:

```
for i in [7, 4, 8, 3]:
    print(i)
```

Tanto en Pascal como en Python, la variable de control puede indicar el número de veces que se repita un bucle, pero no necesariamente: el ejemplo anterior muestra perfectamente que puede indicar los valores que se quieren manejar dentro del bucle (7, 4, 8 y 3) y que son absolutamente independientes del número de iteración.

En Pascal, la variable de control de un bucle `FOR` puede ser de tipo ordinal: entero, carácter, enumerado o subrango (los dos últimos se verán en el Capítulo 8). En Python, puede ser también de otros tipos, como de tipo real.

6.4. Ejercicios

Ejercicio 6. *El factor de fricción de Darcy f de una tubería circular de PVC por la que circula un fluido depende de su diámetro D (en metros), la rugosidad absoluta del material de la tubería K, y el número de Reynolds R_e. Su cálculo se puede realizar, para determinados tipos de flujo, mediante la ecuación no lineal de Colebrook-White, de forma que $f = 1/s^2$, donde s se calcula con la siguiente ecuación recurrente:*

$$s_{i+1} = -2\log_{10}\left(\frac{K}{3.7 \cdot D} + \frac{2.51 \cdot s_i}{R_e}\right)$$

El valor s a usar en el cálculo de la fricción será s_n, que se obtendrá mediante el cálculo iterativo de los valores s_0, s_1 , ..., s_i, $s_{i+1}, \ldots, s_n$ hasta que la diferencia entre los dos últimos valores consecutivos s_n y s_{n-1} sea menor que una tolerancia T, o se haya alcanzado un número máximo de iteraciones $n = N$. Se considerará como valor inicial de la serie $s_0 = K/(3.7 \cdot D)$.

R_e depende de la velocidad del fluido en la tubería ν (m/s), su densidad ρ (kg/m^3), su viscosidad cinemática π (k/m^2) y el diámetro de la tubería D, de acuerdo con la ecuación:

$$R_e = \frac{\rho \cdot \nu \cdot D}{\pi}$$

Escriba un programa Pascal que pida al usuario los valores $D, K, \rho, \nu, \pi, T, N$ y que escriba el valor de f.

Ejercicio 7. *A principios de 2024, un estudiante de la Universidad de Zaragoza usó un sistema de Inteligencia Artificial para resolver un problema: escribir un programa en Pascal que calcule el factorial de un número entero no negativo n. El factorial $n!$ se define como $n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot n$, con $0! = 1$. El código que obtuvo tiene varios errores conceptuales y aspectos susceptibles de mejora, lo que sugiere la importancia de que los humanos continuemos estudiando programación. ¿Sería capaz de detectar estas limitaciones?*

```
PROGRAM factorialChatGPT;
VAR
    entero, n, factorial, factorialant : integer;
BEGIN
    write('Calcula␣el␣factorial␣de:␣');
    readln(entero);
    IF entero = 0 THEN
    BEGIN
        writeln('El␣factorial␣de␣0␣es␣1');
        readln;
    END
    ELSE
    BEGIN
        factorial := 1;
        FOR n := 1 TO entero DO
        BEGIN
            factorial := factorial * n;
        END;
        writeln('El␣factorial␣de␣', entero,
                '␣es:␣', factorial);
        readln;
    END;
END.
```

Ejercicio 8. *"El frutal" es un juego de mesa infantil cooperativo: todos los jugadores humanos colaboran compitiendo contra un cuervo imaginario. La mecánica del juego es la siguiente. Hay un tablero con 4 árboles frutales (un manzano, un peral, un cerezo*

y un ciruelo), cada uno con 10 *piezas de la fruta correspondiente a cada árbol. En el centro del tablero, hay espacio para colocar* 9 *piezas de un puzle. Inicialmente el puzle está vacío y, una vez completado, aparecería el cuervo. El objetivo del juego es retirar la fruta de todos los árboles antes de que se complete el puzle del cuervo. En cada turno, los jugadores humanos tiran un dado de* 6 *caras: una cara por cada tipo de fruta, una cara con el cuervo y una cara especial con una cesta. Si aparece un tipo de fruta, se retira una pieza de fruta de ese tipo del árbol (si queda alguna). Si aparece el cuervo, se coloca una pieza del puzle. Si aparece la cesta, los jugadores retiran dos piezas de fruta elegidas libremente (evidentemente, la mejor estrategia consiste en retirar piezas de los árboles donde queden más frutas por recoger). Es decir, se deben recoger todas las frutas antes de que el cuervo aparezca* 9 *veces al lanzar el dado.*

La ventaja de los juegos cooperativos infantiles es que, aquellos infantes que llevan muy mal perder, no se enfadarán con sus compañeros de juego. Sin embargo, si el cuervo imaginario gana con demasiada frecuencia, el berrinche se lo van a llevar igual. Por ello, es interesante conocer la probabilidad de que los jugadores ganen, y probablemente la manera más rápida de estimarla es hacer un programa que juegue automáticamente al frutal muchas veces (digamos, un millón) y calcule cuántas veces ganan los jugadores. Para simular el lanzamiento del dado, se pueden generar números aleatorios como se discutió al final de la Sección 6.2.

Ejercicio 9. *El código genético de las proteínas está formado por una cadena muy larga de bases nitrogenadas. En el caso del ADN, estas bases son adenina (A), citosina (C), guanina (G) y timina (T). Una terna de estas bases (denominada codón) se corresponde con un aminoácido específico, existiendo además codones especiales para codificar el inicio y el fin de la traducción. En el ADN, el inicio de la región a traducir se codifica con cualquiera de los codones 'TAA', 'TAG' o 'TGA', mientras que el fin de la región a traducir se codifica con los codones 'TTG', 'CTG' o 'ATG'. Una base solamente puede formar parte de un codón.*

Escribir un programa para el procesamiento de código genético de ADN, que deberá leer de teclado una secuencia de bases nitrogenadas, tan larga como se quiera y finalizada con un salto de línea, y escribir por pantalla el número de codones de la primera secuencia de traducción encontrada (es decir, del tramo que abarca desde el primer codón de inicio de la región de traducción hasta el primer codón de fin de la región de traducción, sin contar estos 2 *codones especiales) o bien indicar si no se encontró alguna de estas regiones. Se puede suponer que la entrada no contiene ningún carácter diferente a 'A', 'C', '' y 'T' y que el número de términos en la secuencia es múltiplo de* 3 *(posiblemente* 0*).*

*Por ejemplo, dada la secuencia "CCC**TAA**CCCAAA**TTG**TTG", el programa debe mostrar que se encontraron* 2 *codones y, dada la secuencia "CCC**TAA**CCCAAA", debe indicar que no se encontró el final de la región.*

Ejercicio 10. *Construya un programa que lea de teclado una secuencia de caracteres que represente una fórmula química, tan larga como se quiera, y finalizada con un salto de línea. El programa indicará si es correcta conforme a las condiciones enumeradas a continuación o cuáles de ellas incumple. Una fórmula se considerará correcta si y solo si verifica las siguientes condiciones:*

C1 *Solo contiene números y caracteres (en mayúsculas o minúsculas) del alfabeto inglés.*

C2 *La fórmula comienza con una letra mayúscula.*

C3 *Existen al menos* 2 *letras mayúsculas diferentes entre sí.*

C4 *Una letra minúscula no aparece detrás de otra letra minúscula.*

C5 *Una letra minúscula no aparece detrás de un dígito numérico.*

Por ejemplo, dada la formula química "K2Cr2O7", el programa diría que la fórmula cumple todas las condiciones. En cambio, dada la secuencia de entrada '2-C2rO7', el programa debería decir que la fórmula no cumple las condiciones: "C1 C2 C5".

Ejercicio 11. *Los sulfatos y los cloratos son sales que contienen los aniones SO_4^{2-} y ClO_3^-, respectivamente. Dada una secuencia de caracteres que representan una lista de fórmulas químicas, se puede calcular el número de sulfatos y de cloratos contando el número de apariciones de las sub-secuencias "SO4" y "ClO3", respectivamente.*

Escriba un programa que a partir de una secuencia introducida por teclado de fórmulas químicas, correctamente escritas, y de longitud arbitrariamente larga, escriba por pantalla el número de sulfatos y el número de cloratos encontrados en la secuencia de entrada. Las fórmulas estarán separadas por espacios en blanco. La secuencia finalizará con un salto de línea y contiene al menos 3 *caracteres.*

Por ejemplo, dada la secuencia de formulas "NaCl CuSO4 Na2SO4 NaClO3", el programa debe informar de que se encontraron 2 *sulfatos y* 1 *clorato.*

Ejercicio 12. *La representación simbólica de una reacción química consta de dos partes (izquierda y derecha) separadas por los caracteres (que simulan una flecha) '->' y finaliza con un salto de línea (véase el ejemplo en el cuadro del final del enunciado). Tanto a la izquierda como a la derecha de la flecha, aparecerán uno o más términos.*

Cada término consta de una fórmula de un compuesto químico precedida de un número entero (que, por simplicidad, supondremos que es obligatorio y que siempre consta de un único dígito, posiblemente 1). En el caso de que aparezca más de un término en la parte izquierda o derecha de la fórmula, dos términos consecutivos estarán separados por: (1) un espacio en blanco; (2) el carácter "+"; y (3) otro espacio en blanco.

En este problema, una fórmula química será cualquier secuencia alfanumérica que comienza con una letra mayúscula y finaliza en un espacio en blanco (para facilitar el procesamiento, se añadirá un espacio incluso tras la última fórmula), no siendo necesario comprobar si las secuencias alfabéticas se corresponden con símbolos químicos o no. Además, se supondrá que la reacción está correctamente ajustada.

Escriba un programa que lea de teclado la representación de una reacción química de tamaño tan largo como se quiera de acuerdo con el formato anteriormente descrito, y que muestre por pantalla los distintos términos que aparecen, desglosados uno por cada línea. Se puede suponer que la reacción se introduce correctamente por el usuario.

Por ejemplo, dada la reacción "1 NaOH + 1 HCl ->1 H2O + 1 NaCl" la salida del programa sería:

```
Reactantes:
 1 molecula/s de NaOH
 1 molecula/s de HCl
Productos:
 1 molecula/s de H2O
 1 molecula/s de NaCl
```

Ejercicio 13. *Debido a las restricciones de acceso a tiendas, supermercados y todo tipo de negocios durante la pandemia de COVID-19, se incrementaron considerablemente las colas para poder ingresar a los establecimientos. Una empresa decidió instalar un sistema de videovigilancia para garantizar una distancia de seguridad de al menos* 2 *metros entre las personas que esperan. Concretamente, la tecnología disponible permite obtener una secuencia de valores numéricos, entre* 0 *y* 255*, cada uno de los cuales corresponde al resultado de escanear un área de* $1m^2$*. Valores inferiores a un cierto umbral (*128*) indican que en el área de* $1m^2$ *correspondiente hay una persona, mientras que valores superiores o iguales indican la presencia del suelo.*

Construya un programa que solicite al usuario la escritura por teclado de una secuencia como la descrita anteriormente, de longitud arbitrariamente larga y finalizada con un salto de línea. Como salida, el programa debe mostrar por pantalla un listado de la posición de las personas (el número de área dentro de la secuencia) que incumplen la distancia de seguridad con respecto a la persona de delante, así como el porcentaje de personas que la incumplen.

Por ejemplo, para la secuencia "0 255 255 0 255 0 255 255 0 255 0", la salida es:

```
En el area 6 se incumple la distancia de seguridad
En el area 11 se incumple la distancia de seguridad
Un 40% de personas incumplen la distancia de seguridad
```

Capítulo 7

Subprogramas

Hasta ahora, hemos visto programas relativamente sencillos (al menos, en cuanto al número de líneas de código necesarias para resolver un problema). Por ello, todas las instrucciones del código estaban en el programa principal. Sin embargo, en general, es frecuente que un programa se estructure en varios módulos o subprogramas, especialmente en problemas complejos.

De todos modos, incluso en los programas sencillos que hemos considerado, hemos utilizado subprogramas que son parte del lenguaje de programación, como `writeln` o `sqrt` en Pascal y `print` o `math.sqrt` en Python. Efectivamente, en Pascal para multiplicar x por y se escribe `x * y`, usando un operador de multiplicación, pero para calcular la raíz cuadrada de x se escribe `sqrt(x)`, usando la función `sqrt`.

En este capítulo, aprenderemos mejor el funcionamiento de los subprogramas que existen en el lenguaje y, sobre todo, desarrollemos nuestros propios subprogramas.

7.1. Modularización

Planteémonos por un momento qué sucede cuando ejecutamos `writeln` en Pascal. En una sola línea hemos conseguido nuestro objetivo pero es fácil darse cuenta de que es una operación compleja que, internamente, habrá requerido varias líneas de código. Por un lado, el número de argumentos es variable, y podemos requerir a la función que escriba únicamente un salto de línea o que escriba antes una cadena de caracteres y un número real (como hicimos en la línea 9 de nuestro segundo programa en Pascal). Por otro lado, en caso de que el argumento sea una cadena de caracteres se escriben los caracteres, pero si son números reales se pueden escribir en formato científico o decimal. En definitiva, hemos dividido un problema en varias partes, de modo que una de ellas (la escritura por pantalla) la podemos reutilizar (de hecho, ha sucedido) en cualquier programa que necesite realizar esa tarea. No tenemos ni idea de cómo está implementado por dentro `writeln` pero, ni lo sabemos, ni nos importa, y eso es genial:

lo relevante es que podemos utilizarlo.

Supongamos ahora que queremos el número combinatorio $\binom{n}{k}$, que denota el número de subconjuntos de m elementos que pueden formarse a partir de un conjunto de n elementos, y que tiene múltiples aplicaciones en combinatoria. El número combinatorio puede calcularse a partir del factorial, ya que $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, siendo $n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot n$ el factorial de n, con $0! = 1$ y para todo entero no negativo n. Veamos un ejemplo de código en Pascal para resolver el problema:

```
1  PROGRAM combinatorio;
2  VAR
3      fact, factN, factM, factN_M, i, n, m : integer;
4  BEGIN
5      write('Introducir␣dos␣enteros␣positivos␣n␣y␣m:␣');
6      readln(n, m);
7      fact := 1;
8      FOR i := 2 TO n DO
9          fact := i * fact;
10     factN := fact;
11     fact := 1;
12     FOR i := 2 TO m DO
13         fact := i * fact;
14     factM := fact;
15     fact := 1;
16     FOR i := 2 TO n - m DO
17         fact := i * fact;
18     factN_M := fact;
19     writeln('C(', n, ',␣', m, ')␣=␣',
20         factN DIV (factM * factN_M));
21 END.
```

En este código se repite tres veces un código muy similar, el cálculo del factorial (líneas 7–9, 12–14 y 17–19) donde sólo cambia el valor final del bucle. Claramente, esto es propenso a errores, difícil de mantener, innecesariamente difícil de escribir y de leer. Sería preferible tener una función `factorial` que nos permitiera simplificar el programa principal así:

```
BEGIN
    write('Introducir␣dos␣enteros␣positivos␣n␣y␣m:␣');
    readln(n, m);
    factN := factorial(n);
    factM := factorial(m);
    factN_M := factorial(n-m);
    writeln('C(', n, ',␣', m, ')␣=␣',
        factN DIV (factM * factN_M));
END.
```

Evidentemente, en el código anterior no sería suficiente con modificar el programa principal, sino que habría que incluir la definición de `factorial`. El resultado tendría la siguiente estructura:

```
1  PROGRAM combinatorio;
2  { Factorial }
3  { ... }
4  { Variables del programa principal }
5  VAR
6      { ...}
7  { Programa principal }
8  BEGIN
9      { ... }
10 END.
```

La modularidad tiene numerosas ventajas, entre las que podemos destacar:

- Reusabilidad del módulo para diferentes problemas.
- Simplicidad, mediante descomposición de un programa complejo en partes más sencillas de resolver por separado.
- Abstracción, al permitir usar un módulo como si fuera una instrucción sencilla y sin conocer sus detalles de implementación.
- Legibilidad del código, al evitar repetir instrucciones y permitir la abstracción de detalles.
- Mantenimiento, ya que si fuera necesario modificar el código, solo habría que hacer cambios en un lugar (la definición del módulo) y no todas las veces que se utilice.
- Fiabilidad, ya que las partes de un programa se pueden probar por separado y, una vez que estemos seguros de que un módulo funciona correctamente, los errores en programas que los utilicen deben ser por otros motivos.

El diseño descendente o diseño de arriba a abajo (en inglés, *"top-down"*) es una técnica basada en la idea de "divide y vencerás": descomponer un problema complejo en otros más sencillos. En nuestro caso, se trata de dividir un programa en subprogramas, de menor complejidad y tamaño, y componer de alguna manera los resultados parciales de cada subprograma para calcular el resultado final. A su vez, cada uno de esos subprogramas podría dividirse en otro subprogramas, y el proceso podría repetirse hasta obtener subprogramas de una dificultad asequible.

7.2. Funciones vs. procedimientos

El mecanismo básico para conseguir la modularización son las funciones y procedimientos. Los lenguajes de programas incorporan algunos, pero además proporcionan mecanismos para poder definir nuestras propias funciones, para solucionar casos específicos para los que no existe una función que los resuelva.

En Pascal, se distingue entre funciones y procedimientos. La diferencia es que un procedimiento es un conjunto de instrucciones que se ejecuta cuando el usuario lo invoca pero, al finalizar, no devuelve ningún valor. Por ejemplo, `writeln`. En cambio, una función es un conjunto de instrucciones que se ejecuta cuando el usuario lo invoca y que, al finalizar, devuelve un valor. Por ejemplo, `sqrt` devuelve un valor numérico (el resultado de calcular la raíz cuadrada de un cierto número).

Consideremos las siguientes instrucciones en Pascal, consistentes en dos *llamadas* o usos diferentes de `writeln` y `sqrt`:

```
1 writeln(9);
2 resultado := sqrt(9);
```

En ambas instrucciones se invoca al subprograma incluyendo un parámetro real (9). Ambas instrucciones son absolutamente correctas: la primera (Línea 1) llama a un procedimiento, que no devuelve ningún valor, y la segunda (Línea 2) llama a una función y el valor que devuelve se asigna a una variable para usarlo posteriormente.

Veamos ahora otras dos ejemplos de llamadas, pero incorrectas:

```
1 sqrt(9);
2 resultado := writeln(9);
```

La primera es sintácticamente válida, pero no tiene sentido: se llama a una función, que devuelve un valor, pero no se hace nada con él y, por tanto, se pierde. La segunda es directamente incorrecta desde un punto de vista sintáctico y crearía un error de compilación: como el procedimiento no devuelve ningún valor, no se puede asignar el valor que devuelve a la variable `resultado`.

7.3. Funciones

Un subprograma tiene una *definición* (donde se detalla qué hace). Una vez definido, podemos realizar *llamadas* donde se utilizan. Hasta ahora solo hemos visto llamadas porque los subprogramas que ya existen en el lenguaje no necesitan ser definidos previamente por el programador. Sin embargo, en el caso nuestros subprogramas, tendremos que definirlos una única vez antes de poder usarlos. En esa definición se especificará

qué hace el subprograma dados unos *parámetros formales* de entrada, que serán variables sin un valor conocido. En el momento de la llamada, se indicarán unos *parámetros reales* con un valor conocido.

Veamos un ejemplo de cómo podría implementarse en Pascal una función para calcular raíces cuadradas. Es similar a lo que hace la función `sqrt`, pero recibe un parámetro adicional que especifica la precisión del resultado:

```
1  PROGRAM programaConFunciones;
2
3  FUNCTION raizCuadrada (x : real; epsilon : real) : real;
4  { Asume x >= 0 }
5  VAR
6      r : real;
7  BEGIN
8      r := x;
9      WHILE abs(x - r * r) > epsilon DO
10         r := (r + x / r) / 2;
11     raizCuadrada := r;
12 END;
13
14 VAR
15     num, raiz : real;
16 BEGIN
17    write('Introducir un numero real no negativo: ');
18    readln(num);
19    IF num >= 0 THEN
20    BEGIN
21       raiz := raizCuadrada(num, 1E-10);
22       writeln('Su raiz cuadrada es: ', raiz:1:10);
23    END;
24    writeln('Pulse Intro para continuar...');
25    readln;
26 END.
```

Las líneas 12–24 definen las variables e instrucciones del programa principal, y las líneas 2-11 definen la función para calcular la raíz cuadrada. Como vemos en la línea 2, se trata de una función que recibe dos parámetros formales de entrada (`x`, de tipo real, y `epsilon`, también de tipo real) y devuelve un valor de salida, que es de tipo real. El comentario de la línea 3 también es optativo, aunque es una buena práctica de programación explicar a través de comentarios qué hace cada módulo, que parámetros de entrada necesita, qué dependencias tiene, etc.

Si el usuario ejecuta el programa, se comienza a partir de la línea 15, pidiéndose al usuario que escriba un número real y leyéndolo de teclado (línea 16). Si el número no es negativo (línea 17), se llama a `raizCuadrada` (línea 19). En ese momento, se

interrumpe la ejecución del programa principal y se pasa a ejecutar la función.

En primer lugar, se evalúan los parámetros reales (es decir, los que se suministran en el momento de la llamada, `num` y `1E-10`) y se hacen corresponder con los parámetros formales (`x` y `epsilon`). Es decir, a `x` se asigna el valor de `num` y a `epsilon` el valor `1E-10`. A continuación, se ejecuta el cuerpo de la función. En la línea 7, se asigna a la variable local (propia de la función) `r` el valor de `x`. A continuación (línea 8), se define un bucle que se repite mientras la diferencia entre `x` y `r` al cuadrado sea mayor que la precisión deseada `epsilon`. Esto es equivalente (pero mucho más eficiente desde un punto de vista computacional) a comprobar que la diferencia entre la raíz cuadrada de `x` y `r` es mayor que `epsilon`). En cada iteración del bucle, sustituye el valor de `r` por la expresión `(r + x / r) / 2`, que aproxima la raíz cuadrada de `x` usando el método de Newton-Raphson. Finalmente, cuando ambos valores son suficientemente parecidos, sale del bucle y la función devuelve la aproximación a la raíz cuadrada calculada (línea 10). En Pascal, para definir el valor que devuelve una función, se asigna al nombre de la función (`raizCuadrada`) el valor que queremos que devuelva (`r`).

Una vez que la función ha devuelto el resultado al módulo desde el cual se la llamó (en este caso, al programa principal), se reanuda la ejecución a partir del punto en el que quedó interrumpida (línea 19). Es decir, se asigna a la variable `raiz` el resultado de la llamada a la función. A continuación, se escribe por pantalla el resultado (línea 20). Al igual que en programas anteriores, las líneas 22-23 evitan que se cierre la ventana de ejecución hasta que el usuario pulse la tecla Intro.

Es una buena costumbre de programación, aunque no es obligatorio, que el valor que devuelve una función en Pascal se defina en la última instrucción de la función.

En Pascal las funciones se ejecutan hasta llegar al final (delimitado por el `END`). Aunque antes del final se asigne el valor que queremos que devuelva, eso no hace que la función devuelva ese valor inmediatamente (en Python, sí).

Antes de ver cómo hacer lo mismo en Python, incluyamos una breve mención a la diferencia entre las funciones en informática y matemáticas. En Pascal hemos definido una función `FUNCTION raizCuadrada (x : real; epsilon : real) : real;`. En matemáticas, se habría hecho como $\surd : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Igualmente, tendríamos un "nombre" de función ($\surd$). El dominio de la función (el producto cartesiano de dos reales) podríamos verlo como los parámetros formales de entrada, y determina tanto el número como el tipo de ellos. El codominio o rango sería similar al tipo devuelto por la función, en este caso un un número real. Además, podríamos incluir una llamada a la función proporcionando sus parámetros reales, como $\sqrt{2}$. Sin embargo, en un programa podemos incluir tipos de datos sin un equivalente directo en matemáticas (como caracteres o booleanos) y la definición de la función incluye código (instrucciones) en

vez de una simple expresión matemática.

Ahora consideraremos el caso de Python donde, por defecto, ni los parámetros ni el tipo de salida tienen un tipo estático, pero se puede (como haremos en este texto) forzar usando los *type hints*. En este caso, el código sería el siguiente:

```
1  def raizCuadrada(x: float, epsilon: float) -> float:
2      # Asume s >= 0
3      r: float
4      r = x
5      while abs(x - r * r) > epsilon:
6          r = (r + x / r) / 2
7      return r
8
9
10 num: float
11 raiz: float
12 print("Introducir un número real no negativo: ", end='')
13 num = float(input())
14 if num >= 0:
15     raiz = raizCuadrada(num, 1E-10)
16     print("Su raíz cuadrada es: ", raiz)
```

La función se define en las líneas 1–7 y el programa principal en las líneas 10–16. Por convención, se suelen añadir dos líneas en blanco después de la función (líneas 8–9), aunque es algo voluntario ya que se usa la indentación del código para indicar qué instrucciones van dentro de la función, como en los condicionales y bucles.

La principal diferencia con Pascal es que para especificar que una función devuelve un valor al programa principal se usa la instrucción `return`. Al contrario que en Pascal, en el momento en que se ejecute una instrucción `return` se interrumpe la ejecución del módulo, regresando al módulo que le llamó (en nuestro caso, al programa principal).

Como vemos en la línea 1, tenemos un función que recibe dos parámetros de entrada (`x`, de tipo real, y `epsilon`, también de tipo real) y devuelve un valor de salida, que es de tipo real. El comentario de la línea 2 es optativo, aunque muy recomendable. La línea 3 define una variable local de la función (`r`, de tipo real). Las líneas 4–6 aproximan la raíz cuadrada de `x` usando el método de Newton-Raphson y almacenan el resultado en la variable `r`. Finalmente, la línea 7 devuelve el resultado calculado.

En las líneas 10–11 se definen las variables del programa principal. Cuando el usuario ejecuta el programa, se comienza desde la línea 12, pidiéndole que escriba un número real y leyéndolo de teclado (línea 13). Si este número no es negativo (línea 14), se invoca a la función a la función `raizCuadrada` (línea 15), interrumpiéndose la ejecución del programa principal y pasando a ejecutar la función.

En ese momento, se evalúan los parámetros reales `num` y `1E-10`, se hacen corresponder con los parámetros `x` y `epsilon`, y ejecuta las instrucciones de las líneas 4–6, hasta que la función devuelve la aproximación a la raíz cuadrada calculada (línea 7).

A continuación, se reanuda la ejecución del programa principal a partir del punto en que fue interrumpida por la llamada a función (línea 15) y, tras asignar a la variable `raiz` el resultado devuelto por la función, se escribe el resultado (línea 16).

¡Atención! 10. *Son errores frecuentes:*

- *Pasar un número incorrecto de parámetros en la llamada a un subprograma.*
- *Pasar un parámetro real de distinto tipo que el formal.*
- *Devolver un valor de distinto tipo al de la función (es decir, diferente al tipo de valores de salida especificado en la definición de la función).*
- *Incluir una llamada a función en un expresión con elementos de distinto tipo al tipo del valor devuelto por la función.*
- *Pretender que un subprograma obtenga los valores de entrada necesarios leyendo de teclado (en lugar de usando parámetros de entrada).*
- *Pretender que una función escriba por pantalla un resultado en vez de devolverlo.*

7.4. Procedimientos

Veamos ahora un ejemplo de procedimiento en Pascal. En este caso, el procedimiento recibirá dos parámetros: una cadena de caracteres con el nombre del usuario y un valor booleano que indica su sexo (si es verdadero, será una mujer, y si es falso, será un hombre) y que escribe por pantalla un mensaje personalizado.

```
PROGRAM procedimiento;

PROCEDURE saludo (nombre : STRING; mujer : boolean);
BEGIN
    IF mujer THEN
        writeln('Bienvenida, ', nombre)
    ELSE
        writeln('Bienvenido, ', nombre)
END;

BEGIN
   saludo('Fernando', false);
END.
```

En Python, en realidad, no existen los procedimientos como tales, pero es posible que una función no devuelva ningún valor. Para ello, al definir la función se dice que devuelve un tipo especial que representa la nada (`None`) y no se incluye ninguna sentencia `return` en el cuerpo de la función. El ejemplo anterior quedaría así:

```
def saludo(nombre: str, mujer: bool) -> None:
    if mujer:
        print("Bienvenida,", nombre)
    else:
        print("Bienvenido,", nombre)


saludo("Fernando", false)
```

En Python, también es posible que un módulo (de tipo procedimiento) incluya una instrucción `return` sin ningún valor devuelto; en ese caso se finaliza la ejecución del módulo, regresando al módulo que le llamó pero sin devolverle ningún valor.

7.5. Módulos sin parámetros

Por supuesto, el número de parámetros no es necesariamente dos: podemos tener uno, más de dos e incluso ninguno, aunque esto es poco habitual. Además, no es necesario que todos los parámetros tengan el mismo valor, ni que el valor devuelto por la función coincida con el tipo de alguno de los parámetros.

Los módulos sin parámetros en Pascal se definen e invocan de manera similar a la habitual, solo que la parte entre paréntesis (donde se definen los parámetros) no se incluye. Además, en el momento de la llamada no se incluyen paréntesis. Por ejemplo:

```
PROGRAM procedimiento;

PROCEDURE saludoGenerico;
BEGIN
    writeln('Bienvenido/a')
END;

BEGIN
   saludoGenerico;
END.
```

En Python, en cambio, se incluyen los paréntesis, pero sin contenido:

```
def saludoGenerico() -> None:
    print("Bienvenido/a")


saludoGenerico()
```

Ampliación 10. *Algunos dialectos o compiladores de Pascal permiten incluir los paréntesis sin contenido, como en Python, pero en general no se permite.*

7.6. Parámetros que cambian su valor

Hasta ahora, todos los parámetros que hemos visto eran de entrada. El mecanismo de paso de parámetros es un *paso por valor* o *paso por copia*: el parámetro formal recibe una copia del valor del parámetro real. Si cambia su valor dentro del módulo, se cambia el valor de la copia, pero no el original. Por tanto, los cambios no se ven al finalizar la ejecución del módulo.

Sin embargo, también podemos tener parámetros de entrada/salida, que transmiten un valor de entrada al módulo llamado pero dicho valor cambia a lo largo de la ejecución del módulo y ese cambio cambio se transmite de vuelta al regresar al módulo original. En este caso, el mecanismo de paso por parámetros es un *paso por referencia*, donde se pasa por el propio parámetro (y no una copia). Por tanto, si cambia el valor del parámetro, cambia el valor del parámetro real y el del parámetro formal, pues son el mismo parámetro. Es decir, los cambios dentro del módulo son visibles al salir de él. Cuando un parámetro se pasa por referencia, el parámetro real debe ser una variable (para que, efectivamente, pueda cambiar su valor).

Por ejemplo, veamos un procedimiento que intercambia el valor de dos enteros:

```
PROGRAM pasoReferencia;

PROCEDURE intercambiar(VAR a, b : integer);
VAR
   copiaDeA : integer;
BEGIN
   copiaDeA := a;
   a := b;
   b := copiaDeA;
END;

VAR
   x,  y: integer;
BEGIN
   x := 7;
   y := 15;
   intercambiar(x, y);
   writeln('x:␣', x);
   writeln('y:␣', y);
   readln
END.
```

La salida del programa principal sería *"x: 15* en una línea e *"y: 7"* en otra.

Para entender bien la diferencia entre paso de parámetros por copia y por referencia, puede resultar útil pensar que es lo mismo que sucede con los apuntes. Si un compañero de clase nos pide que le prestemos los apuntes y le damos una fotocopia, si él estropea los apuntes (por ejemplo, derramando un café encima), a nosotros no nos afecta. Si, en cambio, le prestamos nuestra copia de los apuntes, cuando nos los devuelva veremos la mancha en ellos.

En general, por seguridad, se recomienda usar el paso por referencia solamente si queremos que un módulo modifique o pueda modificar el valor de una variable. En el resto de los casos, deberíamos usar paso por copia. Una posible excepción son los parámetros que ocupan mucha memoria (como por ejemplo las matrices de grandes dimensiones), ya que el paso por referencia permite ahorrar el tiempo necesario para hacer una copia.

En Python, el programador no puede elegir entre paso de parámetros por copia y por referencia. Los tipos de datos conocidos como *inmutables*, se pasan por copia; mientras que los tipos de datos *mutables* se pasan por referencia. Los tipos de datos básicos (entero, booleano, real y cadena de caracteres) son inmutables. Las listas son un ejemplo de tipo de dato mutable. Para simular el comportamiento del programa anterior; podríamos diseñar un procedimiento que, en vez de recibir dos variables de entrada/salida, recibiera una lista con dos variables, como en el siguiente ejemplo:

```
def intercambiar(pars: list[int]) -> None:
    copiaDeA: int
    copiaDeA = pars[0]
    pars[0] = pars[1]
    pars[1] = copiaDeA


x: int
y: int
lista: list[int]
x = 7
y = 15
lista = [x, y]
intercambiar(lista)
x = lista[0]
y = lista[1]
print("x:", x)
print("y:", y)
```

Aunque las listas (`list`) las veremos en profundidad más adelante (Capítulo 9), debemos adelantar que dada una variable `lista`, `lista[0]` devuelve el primer elemento

de la lista (en el caso anterior, el valor asociado a la variable `x`) y `lista[1]` el segundo elemento de la lista (es decir, el valor de la variable `y`).

¡Atención! 11. *Son errores frecuentes:*

- *Intentar modificar el valor de un parámetro pasado por copia.*
- *Pasar por referencia algo diferente a una variable*

7.7. Módulos con más de una salida

Pasar definir módulos que devuelven más de un valor, es preciso usar tipos de datos no básicos, que consideraremos en detalle en los Capítulos 8– 9. En Pascal, en general, se requiere utilizar un registro, que es un tipo de dato que permite agregar varias variables de distintos tipos. De ese modo, el módulo devolvería un valor, pero de tipo registro, y ese valor único agregaría todos los valores de salida. En el caso concreto de que todos los valores que devuelva el subprograma sean del mismo tipo, se podría devolver un array en vez de un registro (como veremos, en Pascal los arrays solo puede almacenar valores de un mismo tipo).

En Python, el equivalente directo de los arrays son las listas, que sí permiten almacenar valores de distintos tipos. Por lo tanto, se puede diseñar un módulo que devuelva una lista que agregaría varios valores:

```
def devolverMasMenosX(x: int) -> list[int]:
    res: list[int]
    res = [x, -x]
    return res


lista: list[int]
lista = devolverMasMenosX(5)
print(lista[0], lista[1])
```

7.8. Ámbito de los elementos

El *ámbito* de un objeto de un programa (una constante, una variable, un subprograma, un tipo de dato definido por el programador, etc.) determina en qué partes del programa está definido (es decir, si hay un definición y si tiene vigencia) y, por tanto, puede utilizarse.

En Pascal y Python el ámbito abarca el subprograma donde está su definición y todos sus subprogramas. Un objeto *local* a un subprograma está definido en él mismo. Un objeto *global*, en cambio, está definido en un nivel superior, pero es accesible desde

el subprograma. Las variables usadas en un subprograma pueden ser de uno de estos tipos: locales, globales o parámetros. En el siguiente código, la función `f` accede a una variable de cada uno de estos tipos:

```
PROGRAM ambito;
VAR
    global : integer;

    FUNCTION f ( parametro : integer ) : integer;
    VAR
       local : integer;
    BEGIN
       local := 2 * parametro;
       global := 0;
       f := local + global;
    END;

BEGIN
    writeln('f devuelve el valor: ', f(5));
END.
```

El uso de las variables globales entraña ciertos riesgos. Por un lado, este programa no compilaría si la función `f` se usa junto a un programa principal que no tenga definida una variable llamada `global`, lo que dificulta la reutilización de la función, que es precisamente uno de los objetivos de los subprogramas. Por otro lado, la función modifica el valor de una variable del programa principal, lo que puede tener efectos no deseados en nuestros programas.

¡Atención! 12. *Se recomienda como norma general no usar variables globales (al menos mientras se aprende a programar): si un subprograma necesita acceder a una variable para inspeccionar su valor, debería pasarse como parámetro de entrada y, si necesita acceder a ella para modificar su valor, debería pasarse por referencia. Si en la llamada a la función se debe pasar como parámetro la variable global, el programador es consciente de su uso dentro de la función y, si además se pasa por referencia, de la posibilidad de que cambie su valor.*

No puede haber una variable local llamada igual que el parámetro formal. Sin embargo, sí puede haber una variable global que se llame igual que una variable local o que un parámetro. Conscientemente, hemos querido evitarlo en todos los ejemplos que hemos visto hasta ahora, pero, ¿qué pasaría en esos casos? Tanto en Pascal como en Python (y como en el matrimonio), primero se busca dentro y, lo que no se encuentra dentro, se busca fuera. Por tanto, en caso de haber una variable local con el mismo

nombre que otra variable global, cuando desde un subprograma se usa el identificador de las variables homónimas, se accederá realmente a la variable local.

Definir una variable local con el mismo identificador que otra del programa principal tiene el efecto de *ocultarla*: desde dentro de la función ya no se puede acceder a la variable del programa principal.

Veamos un ejemplo:

```
1  PROGRAM factorialRecursivo;
2  VAR
3      global, n : integer;
4
5      FUNCTION f ( n : integer ) : integer;
6      BEGIN
7          f := n + global;
8      END;
9
10 BEGIN
11     n := 1;
12     global := 2;
13     writeln(f(3));
14 END.
```

En la línea 7 se intenta acceder a la variable `n`, así que se busca "dentro" (en el nivel actual donde se invoca, la función `f`). Como está definida en `f`, detiene la búsqueda e ignora la `n` del programa principal. Por tanto, en la línea 7, el valor de `n` es 14 (debido a la llamada de la línea 13). En cambio, en esa misma línea 7 se intenta acceder a la variable `global` que, como no se encuentra en `f`, se busca (y se encuentra) en el programa principal. Por tanto, el programa principal escribiría por pantalla *"5"*.

7.9. Recursividad

Un subprograma recursivo es un subprograma que en su código contiene una instrucción en la que se invoca a sí mismo. Por supuesto, debe haber un criterio para efectuar la llamada recursiva y para no hacerlo, para evitar que se produzcan infinitas llamadas recursivas. El buscador Google proporciona un resultado divertido si se busca *"recursion"* (en inglés, recursividad): te pregunta si *"Quizás quisiste decir: recursion"*.

Hay quien dice que la iteración es humana y la recursividad, divina. Lo cierto es que la recursividad puede ser muy elegante, pero también muy compleja. En un curso de iniciación a la programación puede ser preferible no intentar desarrollar subprogramas recursivos, pero existen razones para considerar este aspecto en el texto. Por un lado, puede ayudar a entender cómo funcionan el flujo de ejecución cuando se tienen sub-

programas. Por otro, puede evitar a que los programadores hagan llamadas recursivas de manera accidental (uno ha visto varios casos). Finalmente, existen problemas que por su propia naturaleza tienen una solución recursiva inmediata.

Uno de estos problemas es el cálculo del número factorial. Recordemos que, dado un número entero no negativo n, el factorial de n se define como $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots 1$, con $0! = 1$. Claramente, $n! = n \cdot (n-1)!$ si $n > 0$.

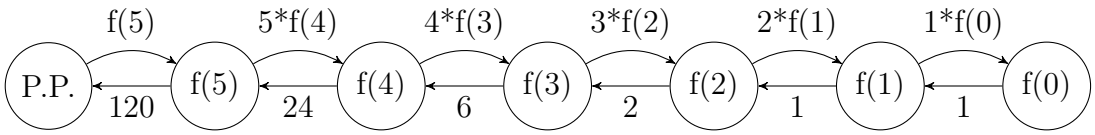
Ya vimos una solución iterativa al principio de este capítulo. En cambio, una posible solución recursiva sería:

```
1  PROGRAM factorialRecursivo;
2  VAR
3      n : integer;
4
5      FUNCTION factorial ( num : integer ) : integer;
6      BEGIN
7          IF num = 0 THEN
8              factorial := 1
9          ELSE
10             factorial := num * factorial(num-1);
11     END;
12
13 BEGIN
14     write('Introducir␣un␣entero␣no␣negativo:␣');
15     read(n);
16     IF n >= 0 THEN
17         writeln(n, '!␣=␣', factorial(n) );
18 END.
```

La función `factorial` comprueba si se ha llegado al caso base, en que `num = 0`. Si es así, se devuelve el valor 1. En otro caso, se devuelve `factorial(num-1)`.

Supongamos que intentamos calcular el factorial de 5. En la línea 8, se intentaría calcular `5 * factorial(4)` para devolverlo como resultado, pero al encontrarse la llamada a `factorial(4)`, el flujo de ejecución se modificaría para atender esa llamada a función. Ahora, se intentaría calcular `4 * factorial(3)` pero, nuevamente, aparecería una llamada a función que se atendería de inmediato. La nueva llamada intentaría calcular `3 * factorial(2)`, que a su vez causaría una llamada que intentaría calcular `2 * factorial(1)`, que produciría una llamada que intentaría calcular `1 * factorial(0)`. Ahora, la llamada a `factorial(0)` devolvería 1 (por ser el caso base) a quien la llamó, que fue la función `factorial` cuando el parámetro `num = 1`. A su vez, esa función puede devolver el valor 1 * 1 = 1 a quien la llamó (`factorial` con el parámetro `num = 2`). A su vez, al recibir ese valor devolvería 2 * 1 = 2 a quien la llamó, que devolvería 3 * 2 = 6 a quien la llamó, que devolvería 4 * 6 = 24 a quien la llamó (la función

`factorial` cuando el parámetro `num = 5`). Finalmente, esta función puede calcular el valor de salida 5 * 24 = 120 y devolverlo a programa principal, que lo escribiría por pantalla. Como, dicho así, parece más un trabalenguas que otra cosa, este proceso se ilustra en el siguiente gráfico donde, por razones de espacio, "P.P." denota el programa principal y, `f` denota a la función `factorial`:



Es muy importante asegurarse en el módulo que invoca a `factorial` de que `n >= 0` porque, si se llama a la función con un parámetro negativo, habría problemas para alcanzar el caso base. Por ejemplo, para -1, se llamaría al cálculo del factorial de -2, que llamaría al cálculo del factorial de -3, y así sucesivamente hasta producirse un desbordamiento (al tratar de acceder a un entero menor que el mínimo representable).

Para acabar esta sección, veamos una versión recursiva del cálculo del número factorial en Python:

```
def factorial(num: int) -> int:
    if num == 0:
        return 1
    else:
        return num * factorial(num - 1)


n: int
print("Introducir un número real no negativo: ", end='')
n = int(input())
if n >= 0:
    print("Factorial: ", factorial(n))
```

7.10. Ejercicios

Ejercicio 14. *En el juego del mus, las dobles parejas (llamadas* duples*) son más valiosas que los tríos (conocidos como* medias*). Por tanto, es de esperar que sean estadísticamente más difíciles de conseguir.*

En la variante más extendida del juego del mus, los doses se consideran ases (o pitos*) y los treses se consideran reyes (o* cerdos*), por lo que se juega con* 8 *pitos y* 8 *reyes. En ciertas zonas del norte de España (País Vasco, Navarra y La Rioja), en cambio, se juega únicamente con* 4 *reyes y* 4 *pitos. ¿Verifican ambas variantes del juego que la probabilidad de obtener duples es menor que la probabilidad de obtener medias?*

Para contestar, resuelva las siguientes tareas:

- *Desarrolle una función que reciba* 2 *cartas de la baraja española y calcule si tienen el mismo valor para el juego del mus o no. Es decir, un tres y un rey tienen el mismo valor, y un as y un dos también.*
- *Desarrolle una función que reciba* 4 *cartas de la baraja española y calcule si las cartas forman duples. Desarrolle otra función para calcular si las cartas forman trío. Tenga en cuenta que tener cuatro cartas iguales no cuenta como trío.*
- *Desarrolle un programa que calcule la probabilidad de que, al repartir* 4 *cartas de una baraja española donde los doses se consideran ases y los treses se consideran reyes, se obtengan* (i) *un trío y* (ii) *una doble pareja.*
- *Adapte los apartados anteriores para que ni los doses se consideren ases ni los treses se consideren reyes.*

En los Capítulos 8–9 veremos cómo definir tipos de datos estructurados (como una carta con un valor y un palo) y listas de datos (como una baraja de cartas). De momento, lo que podemos hacer es representar las cartas de la baraja únicamente con un valor numérico: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$ *denotan los oros (as, dos, tres, cuatro, cinco, seis, siete, sota, caballo y rey, respectivamente),* 11–20 *denotan las copas,* 21–30 *las espadas y* 31–40 *los bastos.*

Ejercicio 15. *Aunque en el mus las dobles parejas son más valiosas que los tríos, en el juego del póker, tener tres cartas iguales (un trío) es una jugada más valiosa que una doble pareja. ¿Es esto coherente con la probabilidad de obtener cada jugada? Para contestar, desarrolle un programa que calcule la probabilidad de que al repartir* 5 *cartas de una baraja francesa sin comodines se obtenga* (i) *un trío y* (ii) *una doble pareja.*

Capítulo 8

Tipos de datos no predefinidos

Los tipos de datos predefinidos en un lenguaje suelen ser insuficientes para resolver numerosos problemas del mundo real. Por ello, los lenguajes de programación incluyen diferentes mecanismos para que el programador pueda definir sus propios tipos de datos, posiblemente a partir de tipos de datos previamente definidos.

Los mecanismos típicos son:

- Renombrado de un tipo definido
- Enumeración de los posibles valores
- Subrango de un tipo definido
- Agregación de tipos en un registro
- Indexación de múltiples valores

En este capítulo, estudiaremos los cuatro primeros. El quinto, la indexación, lo dejaremos para el Capítulo 9.

En Pascal, al igual que existe una zona de declaración de variables (que comienza por `VAR`) y otra de constantes (que comienza por `CONST`), existe una zona de definición de tipos (que comienza con la palabra reservada `TYPE`). Se pueden definir tipos dentro de un programa principal o de un subprograma. Lo lógico es que la definición de tipos se ubique antes de la zona de declaración de variables, para que se puedan definir variables de estos tipos, pero después de la zona de definición de constantes, para que se puedan usar las constantes en las definiciones.

En Python, en cambio, las definiciones de nuevos tipos no tienen una zona concreta donde ubicarse pero, por supuesto, deben aparecer antes de las variables de esos tipos y después de las constantes que utilicen.

8.1. Renombrado

Este mecanismo simplemente permite dar un nombre diferente (un alias) a un tipo que ya existía. No es, por tanto, muy poderoso desde el punto de vista de la expresividad, pero sí que facilita una mejora de la legibilidad de los programas y, en ocasiones, la realización de futuros cambios.

Veamos un ejemplo de cómo hacer esto en Pascal:

```
TYPE
    tipoNumero = real;
```

Veamos ahora cómo hacer lo mismo en Python:

```
tipoNumero = float
```

Verdaderamente no parece demasiado emocionante pero, si en el futuro quisiéramos representar el tipo `tipoNumero` como un entero y no como un real, solamente tendríamos que modificar una línea de nuestro programa. La alternativa de reemplazar, usando cualquier editor de texto, todas las apariciones de `real` por `integer` (si estamos en Pascal), solamente funciona si realmente queremos cambiar absolutamente todas, lo que no es adecuado si en nuestro programa conviven variables de tipo `tipoNumero` con variables de tipo `real`. Este argumento también sirve para el caso de Python.

Además, el renombrado será especialmente interesante para dar un nombre a tipos que no existen previamente.

Al ser un simple renombrado, las operaciones permitidas y la representación del tipo definido son las mismas que las del tipo de dato original.

8.2. Enumeración

Este mecanismo permite definir un tipo de dato de manera extensiva, es decir, mediante un listado exhaustivo de los posibles valores (del dominio). El listado debe ser finito y no incluir repeticiones.

Veamos cómo definir un tipo de dato en Pascal `tipoColor` enumerando sus posibles valores que, por simplicidad, serán solamente rojo, verde, azul y amarillo (al menos, nos permitiría hacer un programa para jugar el parchís)

```
TYPE
    tipoColor = (rojo, verde, azul, amarillo);
```

Podemos crear variables de ese tipo y asignarles valor:

```
VAR
    miColor : tipoColor;
BEGIN
    miColor := rojo;
END.
```

El equivalente en Python sería:

```
from enum import Enum
tipoColor = Enum('tipoColor',
            ['rojo', 'verde', 'azul', 'amarillo'])
miColor: tipoColor
miColor = tipoColor.rojo
```

Este código escribiría por pantalla el valor de la variable "miColor", es decir, *"tipoColor.rojo"*. A la hora de definir el tipo enumerado, se podría haber dado un nombre diferente al tipo resultante (lo que hay a la izquierda del operador de asignación) y al primer argumento de `Enum` pero, en nuestra opinión, así resulta menos confuso.

¡Atención! 13. *Es importante mencionar un error en algunas versiones de Pycharm, en las cuales se genera un aviso del compilador. Este aviso es incorrecto y está documentado como error de Pycharm, que debe solucionarse en futuras versiones [11].*

Ampliación 11. *Python tiene dos sintaxis para los tipos enumerados: la sintaxis funcional que hemos presentado aquí y otra sintaxis basada en clases. En realidad, la sintaxis basada en clases es la más habitual, pero en coherencia con la decisión de no hablar en este texto de clases y objetos, hemos preferido centrarnos en la sintaxis funcional.*

Obsérvese que en Pascal los valores enumerados son identificadores, mientras que en Python son cadenas de caracteres y por eso van entre comillas.

Python permite leer y escribir directamente valores de un tipo de dato enumerado:

```
from enum import Enum
tipoColor = Enum('tipoColor',
            ['rojo', 'verde', 'azul', 'amarillo'])
miColor: tipoColor
print("Introducir␣color:␣", end='')
miColor = tipoColor[input()]
print(miColor)
```

Por ejemplo, si el usuario teclea "rojo", se escribirá por pantalla *"tipoColor.rojo"*.

En Pascal, no es posible leer de teclado ni escribir por pantalla el valor de una variable de tipo enumeración, por lo que el código equivalente sería bastante más complejo:

```
VAR
    miColor : tipoColor;
    linea : STRING;
BEGIN
    { Entrada }
    write('Introducir color: ');
    readln(linea);
    IF linea = 'rojo' THEN
        miColor := rojo;
    IF linea = 'verde' THEN
        miColor := verde;
    IF linea = 'azul' THEN
        miColor := azul;
    IF linea = 'amarillo' THEN
                        miColor := amarillo;
    { Salida }
    CASE miColor OF
        rojo:           writeln('rojo');
        verde:          writeln('verde');
        azul:           writeln('azul');
        amarillo:       writeln('amarillo');
    END
END.
```

Por ejemplo, si el usuario teclea "rojo", se escribirá por pantalla *"rojo"*.

Para la lectura, se ha leído de teclado una cadena de caracteres y, con varios condicionales, se ha asignado a cada cadena un valor de la enumeración. Hay que reconocer que es un proceso bastante tedioso, pero no se puede usar un condicional de alternativa múltiple porque la variable a evaluar (`linea`) es de tipo cadena de caracteres. Para la escritura, sí se puede usar un condicional de alternativa múltiple para escribir un valor diferente por pantalla según el valor del tipo enumerado.

Con respecto a la representación interna, en ambos lenguajes se asocia un valor numérico a cada valor del tipo enumerado. En Pascal, los valores se enumeran a partir del número 0. Es decir, en el ejemplo anterior, `rojo` se representaría con un 0, `verde` con un 1, azul con un `2` y amarillo con un `3`. En Python, la situación es similar con la única excepción de que los valores se numeran a partir del 1.

Con respecto a los operadores relacionales, en ambos lenguajes se permite usar la igualdad y la desigualdad de dos valores de la enumeración. Sin embargo, en Python no se permite usar directamente los demás operadores de comparación. Si queremos comparar, por ejemplo, si un valor es mayor que otro, lo que podemos hacer es comparar los valores enteros con los que se representan internamente. En Python, para conocer el entero con el que se representa `miColor`, se escribe `miColor.value`. El siguiente ejemplo ilustra la comparación de dos valores enumerados:

```
from enum import Enum
tipoColor = Enum('tipoColor',
            ['rojo', 'verde', 'azul', 'amarillo'])
miColor1: tipoColor
miColor2: tipoColor
miColor1 = tipoColor.rojo
miColor2 = tipoColor.verde
print(miColor1.value > miColor2.value)
```

En Pascal, el valor entero se puede obtener con la función `ord`, al igual que con los caracteres. El siguiente ejemplo ilustra las dos posibles maneras de comparar dos valores de un tipo enumerado: directamente o a través de su representación interna:

```
TYPE
    tipoColor = (rojo, verde, azul, amarillo);
VAR
    miColor1, miColor2 : tipoColor;
BEGIN
    miColor1 := rojo;
    miColor2 := verde;
    writeln(miColor1 > miColor2);
    writeln(ord(miColor1) > ord(miColor2))
END.
```

Otra particularidad interesante de los tipos enumerados es que podemos iterar a lo largo de los posibles valores usando un bucle `for`, de modo que el programador no necesita incrementar el valor de la variable de control. Veamos cómo hacer en Pascal un bucle que recorre todos los posibles colores, desde el rojo hasta el amarillo:

```
TYPE
    tipoColor = (rojo, verde, azul, amarillo);
VAR
    miColor : tipoColor;
BEGIN
    FOR miColor := rojo TO amarillo DO
        writeln('Repitiendo␣bucle␣para␣color␣', ord(miColor));
END.
```

Si no queremos repetir el bucle para todos los colores, es tan sencillo como modificar el color inicial y/o el color final. El color final debe ser "mayor" (en términos de representación interna) que el color inicial para que el bucle se repita alguna vez.

Free Pascal proporciona las funciones `low` y `high`, que devuelven el menor y el mayor valor (respectivamente) de una variable de tipo enumerado que se pase como argumento. Por lo tanto, el programa anterior podría quedar así:

```
TYPE
    tipoColor = (rojo, verde, azul, amarillo);
VAR
    miColor : tipoColor;
BEGIN
    FOR miColor := low(miColor) TO high(miColor) DO
        writeln('Repitiendo bucle para color ', ord(miColor));
END.
```

Un código equivalente en Python sería:

```
from enum import Enum
tipoColor = Enum('tipoColor',
    ['rojo', 'verde', 'azul', 'amarillo'])
for i in list(tipoColor)[0:4]:
    print(i)
```

Es decir, para indicar las posiciones inicial y final de los elementos del tipo enumerador para los que tiene que repetirse el bucle, en nuestro caso `rojo` y `amarillo`, usamos la sintaxis `list(tipoColor)[numInicial, numFinal]`.

Si no queremos iterar sobre todos los valores, sino algunos de ellos, también podríamos definir un subrango del tipo `tipoColor`, como veremos más adelante.

En el caso concreto de que queremos iterar sobre todos los colores, se puede iterar directamente sobre todo el tipo de dato:

```
from enum import Enum
tipoColor = Enum('tipoColor',
            ['rojo', 'verde', 'azul', 'amarillo'])
for i tipoColor:
    print(i)
```

En Pascal, si queremos usar un bucle `WHILE` o `REPEAT`, de manera que sea el programador el responsable de incrementar la variable de control, podemos usar las funciones `succ`(x), que devuelve el sucesor de `x` si existe, o `pred`(x), que devuelve el elemento anterior (predecesor) a `x` si existe.

En Python, en cambio, podríamos obtener el campo `.value` del valor enumerado e incrementarlo o decrementarlo. Por ejemplo,

```
miColor: tipoColor
succ: tipoColor
pred: tipoColor
miColor = tipoColor.verde
succ = tipoColor(miColor.value + 1)
pred = tipoColor(miColor.value - 1)
print(pred, miColor, succ)
```

escribe por pantalla: *"tipoColor.rojo tipoColor.verde tipoColor.azul"*.

8.3. Subrango

El tipo de dato subrango permite definir un subconjunto de valores de un tipo ordinal (carácter, entero o enumeración). Al limitar los valores que una variable puede tomar, el compilador puede encontrar errores si intentamos asignar un valor incorrecto. Además, en Pascal resultarán muy útiles para definir los arrays, que veremos más adelante. En ciertas ocasiones, también podrían servir para optimizar el espacio dedicado a almacenar los valores.

El dominio de posibles valores es, por tanto, un subconjunto (en realidad, un intervalo) del dominio del tipo base y el resto (representación interna, operadores, ...) son iguales que los del tipo base.

En Pascal, se usa la sintaxis `valorInicial..valorFinal`. Obsérvese que los valores inicial y final, que denotan los extremos de un intervalo cerrado, están separados por dos puntos y no por tres, como quizá uno podría esperar. Veamos un ejemplo de cómo definir subrangos de enteros, caracteres y enumeraciones:

```
PROGRAM p;
TYPE
    tipoSubrangoChar = 'a'..'z';
    tipoSubrangoEntero = 1..100;
    tipoColor = (rojo, verde, azul, amarillo);
    tipoSubrangoEnum = rojo..azul;
VAR
   c : tipoSubrangoChar;
   numero : tipoSubrangoEntero;
   color : tipoSubrangoEnum;
BEGIN
   c := 'd';
   numero := 5;
   color := azul;
END.
```

En Python, no tenemos un equivalente del tipo subrango para caracteres ni enteros. En el caso de los tipos enumerados, se puede usar `islice`. Por ejemplo:

```
from enum import Enum
from itertools import islice;
tipoColor = Enum('tipoColor',
            ['rojo', 'verde', 'azul', 'amarillo'])
tipoColor2 = islice(tipoColor, 0, 3);
for i in tipoColor2:
    print(i)
```

Dado un tipo enumerado `tipoEnum`, la sintaxis `islice(tipoEnum, numInicial, numFinal)` devuelve un subrango, donde `numInicial` es el valor ANTERIOR al número con el que se representa internamente el valor inicial y‘`numFinal` es el número con el que se representa internamente el valor final. Por ejemplo, el siguiente ejemplo define el tipo `tipoColor2` como un subrango desde `rojo` hasta `azul`:

8.4. Agregación

El mecanismo de agregación permite definir un tipo de dato compuesto por otros tipos de datos (simples o compuestos) previamente definidos. En vez de manejar varias variables por separado, se agrupan para manejarlas de un modo más organizado.

El tipo de dato resultante de la agregación se denomina registro (o, en ocasiones, estructura). Cada uno de los elementos de un registro se denomina campo o atributo. Desde un punto de vista práctico, un registro debe tener al menos dos campos, porque con menos de ellos no tiene sentido hablar de agregación. Los campos de un registro no tienen por qué ser todos del mismo tipo.

De hecho, tradicionalmente, no todos los campos de un registro son del mismo tipo de dato. En tales casos, se suele usar un array, que veremos más adelante. Sin embargo, es posible tener un registro donde todas las variables son del mismo tipo. Incluso, en lenguajes de programación modernos como Python, se puede tener un array con datos de diferente tipo. Por lo tanto, la diferencia entre registro y array no tiene que ver con eso sino con la manera de acceder a los datos agregados: en el registro, cada variable agregada es accesible a través de un identificador, mientras que en un array cada variable es accesible a través de un índice.

Como ejemplo ilustrativo, definiremos un tipo de dato `tipoEstudiante` que permita agregar la información relevante para un estudiante matriculado en cierta asignatura y que, por simplicidad, supondremos que se compone únicamente de un nombre completo (de tipo cadena de caracteres), un número de identificación (de tipo entero) y una calificación (de tipo real). El número de identificación podría ser un DNI, sin su letra, o el Número de Identificación de Alumno, como el usado en la Universidad de Zaragoza.

Veamos cómo definir el registro anterior en Pascal:

```
TYPE
    tipoEstudiante = RECORD
        nombre : STRING;
        id : integer;
        nota : real;
    END;
```

Como vemos, para definir un registro se usa la palabra clave `RECORD`. A continuación, se definen todos los atributos del registro, de la misma manera que se declaran las variables y, por último, se acaba con la palabra clave `END`.

¡Atención! 14. *Este `END` no concluye ningún `BEGIN` previo.*

Una vez definido el nuevo tipo, podemos crear un variable de ese tipo:

```
VAR
    estudiante : tipoEstudiante;
```

Una vez creada, podemos darle valores, teniendo en cuenta que se usa el operador . para acceder al campo de un registro en Pascal. Para ilustrar un caso concreto, representaremos a Kelly Kapowski, con identificador 123 y una calificación de 9.5:

```
estudiante.nombre := 'Kelly␣Kapowski';
estudiante.id := 123;
estudiante.nota := 9.5;
```

Veamos ahora cómo representar el ejemplo anterior en Python. Para ello, usaremos el tipo `TypeDict` que, realmente, permite representar diccionarios con un conjunto fijo de claves, pero que podemos ver cómo el equivalente a los registros:

```
from typing import TypedDict
tipoEstudiante = TypedDict('tipoEstudiante',
    {'nombre': str, 'id': int, 'nota': float})
e: tipoEstudiante
e = {'nombre': 'Kelly␣Kapowski', 'id': 123, 'nota': 9.5}
```

Es importante mencionar que en Python, al contrario que en Pascal, se pueden definir el valor de todos los campos en una única instrucción. No obstante, no es obligatorio dar valor a todos ellos, e incluso se puede crear un registro sin dar valor a ningún campo:

```
from typing import TypedDict
tipoEstudiante = TypedDict('tipoEstudiante',
    {'nombre': str, 'id': int, 'nota': float})
e: tipoEstudiante
e = { }
```

Ampliación 12. *Al igual que con los tipos enumerados, Python tiene dos sintaxis para los registros: la sintaxis funcional que hemos presentado aquí y otra sintaxis basada en clases.*

Para modificar el valor del campo de un registro ya creado, se usa la siguiente sintaxis:

```
e['nombre'] = 'Kelly␣Kapowski'
e['id'] = 123
e['nota'] = 9.5
```

Ampliación 13. *Como veremos en el Capítulo 9, esta sintaxis considera el registro como un array donde los índices para acceder a las posiciones del array son los nombres de los atributos del registro.*

Ampliación 14. *Otra manera de crear registros en Python es definiendo nuestra propia clase, pero hemos decidido no tratarlo en este texto. Otra alternativa más es usar la clase* `NamedTuple` *(donde para acceder al valor de un atributo se usa el operador* `.`*, como en Pascal). Sin embargo, no lo recomendamos en este texto porque es un tipo inmutable, por lo que en el momento de la creación se debe conocer el valor de todos los atributos y no se puede cambiar el valor de los atributos del registro una vez creado:*

```
import typing
tipoPersona = typing.NamedTuple("tipoPersona",
    nombre=str, id=int, nota=float)
p = tipoPersona(nombre='Kelly␣Kapowski', id=123, nota=9.5)
print(p.nombre)
```

Hablemos ahora de algunos operadores definidos para los registros. Tanto en Pascal como en Python, es posible la asignación de registros. En Pascal es imprescindible que a ambos lados del operador de asignación tengamos exactamente el mismo tipo. Conviene aclarar que dos registros definidos con distinto nombre pero compuestos por los mismos atributos (con el mismo número de atributos y siendo cada atributo del mismo tipo) se consideran diferentes tipos. En Python, no es un problema porque todos los registros se consideran realmente de tipo diccionario. Por ejemplo:

```
from typing import TypedDict
tipoEstudiante = TypedDict('tipoEstudiante',
    {'nombre': str, 'id': int, 'nota': float})
e: tipoEstudiante
e2: tipoEstudiante2
e = {'nombre': 'Kelly␣Kapowski', 'id': 123, 'nota': 9.5}
e2 = e
```

En Python están permitidas la igualdad y desigualdad de registros, pero en Pascal no. El resto de los operadores relacionales no está permitido ni en Pascal ni en Python. Para comprobar si dos registros son iguales, se comprueba si tienen los mismos atributos

(con el mismo nombre) y los valores de todos ellos son iguales entre sí o no. Por ejemplo, en el ejemplo anterior, podríamos añadir el siguiente código, que devolvería *"True"*:

```
print(e == e2)
```

Por último, hablemos de las operaciones de entrada y salida. En Pascal no es posible leer un registro de teclado ni escribir un registro por pantalla: debemos hacer esas operaciones independientemente para cada campo del registro. En Python, en cambio, sí que se permiten hacer ambas operaciones. Para escribir un registro por pantalla, basta usar `print`. Por ejemplo, en el ejemplo anterior,

```
print(e)
```

escribiría por pantalla:

{*'nombre': 'Kelly Kapowski', 'id': 123, 'nota': 9.5*}

Con respecto a la entrada de datos, se aplica la función `eval` al resultado devuelto por `input`. Esencialmente, `eval` permite evaluar una expresión en Python y devuelve su resultado. Por ejemplo, dado el siguiente código

```
from typing import TypedDict
tipoEstudiante = TypedDict('tipoEstudiante',
    {'nombre': str, 'id': int, 'nota': float})
e: tipoEstudiante
e = eval(input())
```

el usuario debería teclear la cadena de texto

{*'nombre': 'Kelly Kapowski', 'id': 123, 'nota': 9.5*}

para representar los datos de nuestra ficticia estudiante Kelly Kapowski.

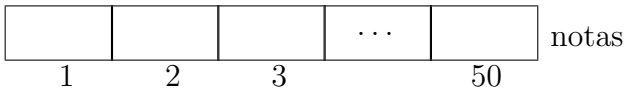
Capítulo 9

Indexación

Si el mecanismo de agregación permite la definición de tipos de datos compuestos por varios datos, cada uno de los cuales es accesible a través de un nombre, el mecanismo de indexación permite la definición de tipos de datos compuestos por varios datos, cada uno de los cuales es accesible a través de un índice. De hecho, es posible considerar varias dimensiones, por lo que habría varios índices (uno por cada dimensión). De momento, comenzaremos considerando el caso unidimensional.

9.1. Creación de vectores unidimensionales

Por ejemplo, supongamos que queremos almacenar las notas de 50 estudiantes. Podríamos usar 50 variables nota1, nota2, ...y nota50, pero esto no permitiría hacer un bucle que recorra todas las notas (por ejemplo, para otorgar un aprobado general asignando a todas ellas un valor de 5). Será preferible usar una variable compuesta `notas`, que permita especificar la primera de las notas como `notas[1]`, la segunda como `notas[2]`, etc. tal y como se muestra en el siguiente diagrama:



Con el nombre de genérico de vectores se suele denominar a la estructura de datos obtenida como indexación de una serie finita de elementos. Estrictamente hablando, se habla de vector cuando tenemos una única dimensión para los índices; si tenemos 2 o más dimensiones, suele usarse el nombre de matriz o tabla. A cada uno de los datos almacenados en el vector se le suele llamar "casilla" del vector.

Normalmente, todos los datos son del mismo tipo. En caso caso, cada vector V hace corresponder a un índice un valor del tipo de datos de los componentes del vector. Formalmente, un vector es una función (en el sentido matemático) $V : I \to D$, siendo I un conjunto de índices y D el tipo de dato de los elementos.

Existen notables diferencias en la forma en la que diferentes lenguajes de programa-

ción manejan los vectores. Con frecuencia, los lenguajes de programación (como Pascal, C o Java) tienen vectores estáticos, de tamaño fijo, aunque también tienen vectores de tamaño dinámico, que no suelen considerarse en los cursos de introducción a la programación. Típicamente, los arrays estáticos se almacenan en posiciones consecutivas de memoria, lo que permite un acceso eficiente a cualquiera de sus casillas.

En Pascal, los vectores de tamaño fijo se llaman *arrays*, a veces traducido al castellano como *arreglos*. Todos los elementos de un array tienen que ser del mismo tipo y los índices pueden ser (un subrango de) números enteros, (un subrango de) caracteres o un tipo enumerado. En cuanto a los índices, Pascal es más flexible que Python (aunque en Python el equivalente al array sería una lista) y otros lenguajes como C o Java, donde los índices siempre son números enteros y a la primera casilla siempre le corresponde la casilla 0.

Consideremos dos ejemplos adicionales de arrays: una estructura para contar la frecuencia de aparición de letras en un texto (lo que tiene aplicaciones, por ejemplo, en criptografía)

17	3	5	···	1
'a'	'b'	'c'		'z'

y otra para almacenar las horas trabajadas durante una semana:

7.5	8.0	8.5	···	0.0
lunes	martes	miercoles		domingo

Estas tres estructuras podrían representarse, respectivamente, con un array de reales con índices enteros, un array de enteros con índices de tipo carácter, y un array de reales con índices de un tipo enumerado `tipoDiasSemana`. Veamos cómo definir los tipos de datos necesarios en Pascal:

```
TYPE
    vectorNotas = ARRAY [1..20] OF real;
    vectorLetras = ARRAY ['a'..'z'] OF integer;
    vectorHoras = ARRAY [ (lunes, martes, miercoles, jueves,
                           viernes, sabado, domingo) ] OF real;
```

Observemos que después de la palabra clave `ARRAY` se indican entre corchetes los índices, desde su valor inicial hasta su valor final. En los dos primeros casos, se está usando un subrango (de los enteros y de los reales), mientras que en el segundo caso se está usando un tipo enumerado. También se podrían haber usado tipos de datos subrango o enumerados previamente definidos por el programador con un nombre:

```
tipoIndice = 1..20;
tipoDato = real;
vectorNotas = ARRAY [tipoIndice] OF tipoDato;
```

Dado que en Pascal el tamaño es fijo, es habitual que no se usen todas las posiciones: el programador define un array del tamaño máximo que podría llegar a necesitar, y además usa una variable para almacenar el tamaño real usado. Digamos que el definir el array se establece el aforo máximo, pero la afluencia puede ser inferior al aforo máximo. Es incluso habitual definir un registro para agregar el array y una variable entera para almacenar su tamaño real. En el siguiente ejemplo, el atributo `tam` permite almacenar tamaños inferior al tamaño máximo (20).

```
vector = RECORD
    v : ARRAY [1..20] OF real;
    tam : 0..20;
END;
```

En Python, la manera más sencilla y habitual Python de obtener un tipo de dato mediante el mecanismo de indexación son las listas [7]:

```
vector = list[float]
```

Nótese que no hemos indicado el tamaño máximo de la lista. El motivo es que en Python las listas no tienen un tamaño estático. Tampoco hemos indicado el tipo de los índices, que siempre es un número entero y la primera posición tiene el índice 0.

En comparación con Pascal, Python es más flexible en cuanto al contenido de una lista (se permite que los valores de las casillas pertenezcan a diferentes tipos de datos) pero menos flexible en cuanto a los índices (ya que son enteros comenzando por cero). Ya que el tipo de los índices no se puede elegir y que cada elemento podría ser de un tipo de dato diferente, en principio no es necesario definir previamente el tipo de dato, pero, como en todo este texto, recomendamos hacerlo aprovechando los *type hints*.

Es algo sorprendente para quienes se inician en el mundo de la programación que las posiciones de las listas comiencen a numerarse partiendo del cero. Un posible motivo es que así sucedía en lenguajes como C o Java, con arrays de tamaño fijo: si cada casilla de la lista ocupa b bytes, las posiciones se numeran a partir de la cero y la lista se almacena en posiciones contiguas de memoria, para acceder a una posición `v[i]` podemos sumar un desplazamiento de $i * b$ bytes a la posición inicial. Como curiosidad, el Gobierno de España también comenzó a numerar a partir de cero las fases de la desescalada de la pandemia de COVID-19.

El operador más importante de los vectores es el que permite acceder a una casilla concreta del vector. En Pascal (y Python), se expresa el índice de la casilla entre corchetes. Por ejemplo, dado un vector `v`, para indicar que el primer estudiante tiene una nota de 4.0:

```
TYPE
  vectorNotas = ARRAY [1..20] OF real;
VAR
  v : vectorNotas;
BEGIN
  v[1] := 4.0;
END.
```

De hecho, en Pascal no existe ningún mecanismo para dar valor a todas las casillas del vector en una única instrucción: debe hacerse casilla a casilla.

Ampliación 15. *Las tuplas de Python son un tipo de dato similar a las listas pero son inmutables, por lo que en general son menos interesantes.*

En Python, a diferencia de Pascal, sí se puede dar valor a todas las casillas de una lista en una única instrucción:

```
vector = list[float]
l: vector
l = [4.0, 7.0, 9.0]
```

La primera línea del código anterior define el tipo de dato. En la segunda, se declara el tipo de la variable `l`. Sin embargo, no se crea realmente ninguna lista hasta la línea tercera, donde se indica implícitamente el tamaño de la lista (3) y su contenido (4.0 en la primera casilla, 7.0 en la segunda y 9.0 en la tercera). Si intentamos usar la lista sin haberla creado, se produce un error en tiempo de ejecución, como se ilustra en el siguiente código erróneo:

```
vector = list[float]
l: vector
print(l)  # NameError: name 'l' is not defined.
```

¡Atención! 15. *Python recomienda no utilizar `l` como nombre de variable por ser "ambigua", ya que se parece demasiado a `I`. Evidentemente, tampoco se recomienda `I` ni `O` (por su similitud con el cero).*

Como el tamaño de una lista es variable, es accesible a través de la función `len`. En el ejemplo anterior, `len(l)` devolvería el entero 3.

En Python, también se usa el corchete para acceder a una posición concreta de una lista para inspeccionar su valor o modificarlo (recuérdese que 0 denota la primera posición). Las listas de Python son objetos mutables, por lo que es posible cambiar el valor de una casilla. Por ejemplo, cambiemos el valor de la primera casilla del vector:

```
l[0] = 5.0
```

En Python no se puede crear una lista con varias casillas sin darles valor a todas ellas. A partir de una lista ya creada (incluso aunque sea una lista vacía), sí que se podría añadir un elemento más con la función `append`. Dada una lista `v` (seguiremos la recomendación de no usar `l`), la instrucción `v.append(x)` amplía el tamaño de `v` en una unidad, añadiendo el elemento `x` al final de la lista:

```
vector = list[float]
v: vector
v = []    # Lista vacía
v.append(10.0)
print(v)  # Escribe: [10.0]
```

En Python, también se permite algo más sofisticado: obtener un subconjunto de una lista ubicado entre una posición inicial (incluida) y otra final (no incluida). En inglés, se usa el término *slicing*, porque se "trocea" una lista:

```
vector = list[float]
v: vector
v = [4, 8, 15, 16, 23, 42]
print(v[2:4])  # Escribe: [15, 16]
```

En Pascal, los vectores tienen pocas operaciones definidas. Es posible asignar directamente un vector a otra variable del mismo tipo pero, en general, no es posible comparar dos vectores usando operadores relacionales (excepto para vectores de caracteres). Tampoco se puede usar `read`/`readln` para leer un vector de teclado, ni `write`/`writeln` para escribirlo por pantalla: estas operaciones deben hacerse elemento a elemento, como veremos a continuación.

9.2. Iteración indexada

Para recorrer todas las casillas de un vector, para realizar las operaciones que deben hacerse elemento a elemento (inspeccionando o modificando sus datos), se utiliza una iteración indexada. Para ello, se usa un bucle que recorre los índices del vector y, a partir de esos índices, se accede a las casillas. Por ejemplo, veamos como dar un aprobado general:

```
PROGRAM aprobadoGeneral;
TYPE
  vectorNotas = ARRAY [1..20] OF real;
VAR
  i : integer;
  v : vectorNotas;
BEGIN
```

```
  FOR i := 1 TO 20 DO
     v[i] := 5.0;
END.
```

Típicamente, se desea recorrer exactamente todas las casillas del vector y se utiliza un bucle definido `FOR` con una variable de control del mismo tipo que los índices del vector (en este caso, de tipo entero). En ciertas ocasiones, se desea recorrer casillas según otra condición de parada y se usa un bucle indefinido.

Dado que en Pascal los índices de un vector pueden comenzar a numerarse en distintos números, Free Pascal proporciona las funciones `low(v)` y `high(v)`, que devuelven el menor y el mayor (respectivamente) de los posibles índices de un vector `v`. Recordemos que estas funciones también están definidas para los tipos enumerados.

En Python, el equivalente sería el siguiente:

```
for i in range(len(l)):
    l[i] = 5.0
```

Recordemos que `range(x)` es equivalente a `range(0, x)`, que devuelve una lista con los valores enteros en $[0, x)$.

Una posibilidad muy interesante de Python, sobre todo para índices no numéricos (Sección 9.6), es que es posible iterar sobre todos los elementos del vector sin acceder a sus índices. Es decir, en vez de usar `for i in range(len(l)):` usaríamos algo como `for dato in l:` cuando quisiéramos inspeccionar el contenido de un vector pero no para modificarlo.

Por último, la función `enumerate` permite iterar sobre un vector obteniendo al mismo tiempo el índice y el dato, por lo que podríamos usar los dos o el que nos interese de los dos. La expresión `for i, dato in enumerate(v):` devuelve tanto el índice de cada casilla (`i`) como el valor almacenado (`dato`).

Veremos un ejemplo de estas dos últimas posibilidades en la siguiente sección, para mostrar el contenido de un vector por pantalla.

9.3. Lectura y escritura de vectores unidimensionales

Dado que, en general, desconoceremos a priori el valor inicial de cada casilla del vector, es frecuente solicitar al usuario que escriba el contenido de cada casilla usando el teclado. Dado que es una tarea frecuente, crearemos un procedimiento para ello, que recibirá como parámetro por referencia (pues cambiará su contenido) el vector y el tamaño del vector:

```
PROCEDURE leerVector(VAR v : vector; tam : integer);
VAR
   i : integer;
BEGIN
   FOR i := 1 TO tam DO
   BEGIN
      write('Escriba␣valor␣en␣la␣posicion␣', i, ':␣');
      readln(v[i]);
   END
END;
```

En este procedimiento hay varias suposiciones implícitas:

- `vector` tiene índices de tipo entero.
- El valor de `tam` es menor que el tamaño máximo usado en la definición del tipo de dato `vector`.
- El tipo de dato de los elementos del vector es un tipo que se puede leer directamente usando `readln` (y, un poco más adelante, supondremos también que se puede escribir con `writeln`). Es decir, podrían ser caracteres, enteros o reales, pero no registros.

Veamos ahora cómo hacerlo en Python:

```
def leerVector(tam: int) -> vector:
    i: int
    v: vector
    v = []
    dato: float
    for i in range(tam):
        print("Escriba␣valor␣en␣la␣posición", i, ":␣", end='')
        dato = float(input())
        v.append(dato)
    return v
```

Ampliación 16. *Hemos optado por crear una función en vez de un procedimiento porque, dado que no se puede crear una lista sin especificar sus valores, en caso de querer pasar como parámetro una lista, en el programa principal que invoque a* `leerVector` *debería crearse una lista vacía, lo que quizá no tiene mucho sentido.*

También es frecuente visualizar el contenido de un vector por pantalla, tal y como se ilustra con el siguiente procedimiento de Pascal:

```
PROCEDURE escribirVector(v : vector; tam : integer);
VAR
   i : integer
BEGIN
   FOR i := 1 TO tam DO
      write(v[i], '␣');
END;
```

En Python, el equivalente sería bastante similar. La principal diferencia es que no es necesario transmitir como parámetro de entrada el tamaño real de la lista, ya que se puede obtener usando la función `len`:

```
def escribirVector(v:vector) -> None:
    i: int
    for i in range(len(v)):
        print(v[i], "␣", end='')
```

También podría, directamente escribirse la lista usando `print`. El inconveniente es que es menos adecuado para modificar el formato por defecto (valores separados por comas).

```
def escribirVector(v:vector) -> None:
    print(v)
```

Veamos ahora una versión iterando sobre todos los elementos del vector sin acceder a sus índices:

```
def escribirVector(v:vector) -> None:
    dato: float
    for dato in v:
        print(dato, "␣", end='')
```

Para concluir esta sección, veamos un ejemplo obteniendo tanto el índice como el valor de cada casilla:

```
for i, dato in enumerate(v):
    print(i, ":␣", dato)
```

9.4. Vectores multidimensionales

Hasta el momento nos hemos restringido a vectores unidimensionales pero, tal y como ya se avanzó, es posible tener vectores unidimensionales, que suelen llamarse matrices. El ejemplo más habitual son las matrices bidimensionales, aunque los vectores se pueden generalizar a más de dos dimensiones.

En Pascal, la declaración de una matriz debe especificar el tipo de los índices para cada una de las dimensiones, además del tipo de dato de cada casilla (que es único para todas). Los tipos de los índices se indican entre corchetes, separando por comas los tipos de cada dimensión. Recordamos que el tipo de los índices no tiene por qué ser numérico. Por ejemplo, para definir una matriz 3×3 de números enteros:

```
matriz =  ARRAY [1..3, 1..3] OF integer;
```

Una matriz bidimensional de reales también puede verse como un array unidimensional donde cada casilla contiene un array unidimensional de reales. En coherencia con ello, otra manera equivalente de definir la matriz anterior sería:

```
matriz =  ARRAY [1..3] OF ARRAY [1..3] OF integer;
```

Para definir una matriz que permita representar un tablero de ajedrez, en cambio, escribiríamos:

```
tipoTableroAjedrez =  ARRAY ['a'..'h', 1..8] OF tipoCasilla;
```

Esto supone que se ha definido previamente el tipo `tipoCasilla`, lo que podría hacerse de la siguiente manera:

```
tipoNompre = (peon, caballo, alfil, torre, dama, rey);
tipoColor = (blanco, negro);
tipoPieza = RECORD
   tipo: tipoNompre;
   color: tipoColor;
END;
tipoCasilla = RECORD
   libre: boolean;
   pieza: tipoPieza;
END;
```

Es decir, cada casilla puede estar libre o no estarlo. Si es falso que esté libre, se indicaría qué pieza ocupa la casilla, especificando su tipo y su color. Observemos que los nombres de las piezas están definidos en orden creciente de valor.

Al igual que con los vectores, es habitual utilizar en la práctica menos casillas que el tamaño máximo especificado en la definición del tipo de dato matriz. Esto hace que, además del array multidimensional, se usen varias variables (una por dimensión) representando el tamaño real: el número de filas reales, el número de columnas reales, etc.

También es habitual definir la matriz como un tipo estructurado que agrega la propia matriz (el array multidimensional) y las variables que almacenan el tamaño real utilizado:

```
matriz = RECORD
   m : ARRAY [1..3, 1..3] OF integer;
   filas, columnas : 0..3;
END;
```

Al definir una lista similar en Python, el tipo sería una lista de listas de tipo entero. Recordemos que los índices no se especifican por que siempre son números enteros comenzando en el cero. Al crear la lista, se puede directamente especificar los valores de sus casillas:

```
matriz = list[list[int]]
m: matriz
m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Para acceder a una casilla de una matriz `m` en Pascal, podemos especificar entre corchetes las coordenadas de cada casilla, separadas por comas:

```
m[1,1] := 0
```

Alternativamente se puede usar una pareja de corchetes para cada dimensión, lo que sería coherente con la visión de una matriz como un array de arrays:

```
m[1][1] := 0
```

En Python, se especifica una pareja de corchetes para cada dimensión:

```
m[0][0] = 0
```

Para acceder a todas las casillas de una matriz, se realiza una iteración indexada, pero con tantos bucles *anidados* como dimensiones tenga la matriz. Por ejemplo, veamos cómo se leería una matriz bidimensional en Pascal:

```
PROCEDURE leerMatriz(VAR m : matriz; int tam : integer);
VAR
    i, j : integer;
BEGIN
    FOR i := 1 TO tam DO
        FOR j := 1 TO tam DO
        BEGIN
            write('Escriba␣valor␣en␣la␣posición␣',
                    i, ',␣', j, ':␣');
            readln(m[i,j]);
        END
END;
```

La variable `i` almacena el índice de la primera dimensión (las filas), mientras que la variable `j` almacena el valor de la segunda (los columnas). Para cada fila `i`, se debe acceder a cada columna `j`.

El equivalente en Python sería algo diferente ya que, al crear la lista `v`, está vacía, por lo que no se puede acceder a una casilla, sino que se debe usar `append` para añadir un nuevo dato:

```
def leerMatriz(tam: int) -> matriz:
    i: int
    j: int
    lista: matriz
    lista = []
    dato: float
    for i in range(tam):
        lista.append([])
        for j in range(tam):
            print("Escriba␣valor␣en␣la␣posición␣",
                i, ",", j, ":␣", end='')
            dato = float(input())
            lista[i].append(dato)
    return lista
```

Además, en vez de pasar la lista por referencia, hemos optado por hacer que la función devuelva la lista.

Consideremos ahora la escritura de una matriz:

```
PROCEDURE escribirMatriz(m : matriz; int tam : integer);
VAR
    i, j : integer;
BEGIN
    FOR i := 1 TO tam DO
    BEGIN
        FOR j := 1 TO tam DO
            write(m[i,j], '␣');
        writeln;
    END
END;
```

El principal aspecto destacable es que en el bucle interno (para diferentes valores de `j`) se usa `write`, para que todas las columnas de la fila actual se escriban en la misma línea, separadas por espacios en blanco (sería preferible en realidad usar tabuladores). Sin embargo, al finalizar cada iteración del bucle externo (para diferentes valores de `i`) se escribe un salto de línea, para que la siguiente fila se escriba en una línea diferente.

El equivalente en Python sería el siguiente, donde se usa `end=' '` para que las columnas se separen por un espacio en blanco:

```
def escribirMatriz(m: matriz) -> None:
    i: int
    j: int
    for i in range(len(m)):
        for j in range(len(m)):
            print(m[i][j], end='␣')
        print()
```

Los códigos anteriores para la lectura y la escritura asumen que las matrices son cuadradas. En el caso de no ser así, el código en Pascal debería adaptarse para recibir dos parámetros: el número de filas usadas y el número de columnas usadas (o, en el caso de matrices multidimensionales, el tamaño real de cada dimensión). En Python, el número real de filas y columnas de una matriz `m` se calcula como

```
numFilas = len(m)
numCols = len(m[0])
```

Si vemos una matriz como una lista de listas, `m[0]` sería la lista que representa la primera fila, así que `len(m[0])` sería la longitud de dicha lista, es decir, el número de columnas. En el caso de que cada fila pueda tener un número diferente de columnas, habría que calcular `len(m[i])` para cada columna i.

9.5. Vectores de registros

Crear vectores de registros en Pascal no tiene misterio, se definen exactamente igual que para los tipos de datos predefinidos. La única observación necesaria es que, a la hora de leer un vector de teclado o escribirlo por pantalla, `read` y `write` no saben manejar registros, por lo que las operaciones deben hacerse para cada atributo del registro. Veamos primero cómo definir los tipos de datos necesarios (el registro y el vector):

```
PROGRAM vectorRegistros;
TYPE
    tipoEstudiante = RECORD
        nombre : STRING;
        id : integer;
        nota : real;
    END;
    vector = ARRAY[1..20] OF tipoEstudiante;
```

A continuación, creamos un array con tres registros representando tres estudiantes: Kelly Kapowski, Zack Morris y Screech Powers:

```
VAR
    v : vector;
    e : tipoEstudiante;
BEGIN
    e.nombre := 'Kelly Kapowski';
    e.id := 123;
    e.nota := 9.5;
    v[1] := e;
    e.nombre := 'Zack Morris';
    e.id := 456;
    e.nota := 5.0;
    v[2] := e;
    v[3].nombre := 'Screech Powers';
    v[3].id := 789;
    v[3].nota := 4.0;
END.
```

Para los dos primeros estudiantes (Kelly y Zack), se ha usado una variable de tipo registro (`e`) y después se ha asignado esa variable a una casilla del vector. Para el tercer estudiante (Screech), se ha considerado directamente que `v[3]` denota un registro, el tercero del vector, y se ha accedido a sus tres campos `nombre`, `id` y `nota`. Por supuesto, se puede (y es lo lógico) usar una misma manera para manejar todos los estudiantes; aquí se ha hecho de un modo diferente por motivos didácticos.

En Python, para crear el equivalente a los vectores de registros, consideraremos listas de `TypedDict`, ya usado en la Sección 8.4. La sintaxis de `TypedDict` es algo más complicada que en Pascal. Veamos cómo sería ahora la lista de tres estudiantes:

```
from typing import TypedDict
tipoEstudiante = TypedDict('tipoEstudiante',
    {'nombre': str, 'id': int, 'nota': float})
e: tipoEstudiante
v: list[tipoEstudiante]
e = {'nombre': 'Kelly Kapowski', 'id': 123, 'nota': 9.5}
v = [e, {'name': 'Zack Morris', 'id': 456, 'nota' : 5.0} ];
v.append({'name': 'Screech Powers', 'id': 789, 'nota' : 4.0});
```

Cada registro se ha manejado de una manera diferente para ilustrar diferentes posibilidades. El primero, correspondiente a la estudiante Kelly, se ha almacenado previamente en una variable `e`. El segundo, correspondiente al estudiante Zack, se ha definido directamente en la misma línea en que se creaba la lista. El tercero, correspondiente al estudiante Screech, se ha añadido a la lista después de su creación.

Al ser objetos mutables, podemos modificar los campos de cualquier registro. Por ejemplo, para aprobar a Screech:

```
v[2]['nota'] = 5.0;
```

9.6. Vectores con índices no numéricos

Ya hemos visto que en Pascal es posible que los índices de los arrays no sean números enteros. Puesto que en Python los índices de las listas siempre son números enteros, para usar índices no numéricos debemos usar un tipo de dato diferente. Concretamente, usaremos diccionarios, que son estructuras de datos que permiten almacenar pares de la forma clave-valor.

Comencemos definiendo un array con índices numéricos comenzando en el uno:

```
v: dict[int, str]
v = {}
v[1] = 'f'
v[2] = 'e'
v[3] = 'r'
v[4] = 'n'
v[5] = 'a'
v[6] = 'n'
v[7] = 'd'
v[8] = 'o'
```

En la primera línea, se ha creado un vector cuyos índices son de tipo entero y donde en cada casilla hay una cadena de caracteres. En la segunda línea, se ha creado un diccionario. En las siguiente casillas, se han insertado caracteres en cada una de las posiciones (entre la 1 y la 8). Observemos que, a diferencia de las listas, se ha podido acceder directamente a las posiciones sin haberlas definido previamente y sin tener que usar la función `append` para añadirlas al final.

Una manera equivalente de expresar lo anterior sería la siguiente:

```
v: dict[int, str]
v = {1: 'f', 2: 'e', 3: 'r', 4: 'n', 5: 'a', 6: 'n', 7: 'd',
     8: 'o'}
```

Es decir, al crear diccionario se especifica el contenido de las casillas, accedidas a través de su índice de tipo entero. Nótese que en este caso no se han usado los corchetes.

Veamos ahora cómo crear un diccionario cuyos índices son caracteres (o cadenas de caracteres) y donde cada casilla contiene un número entero:

```
v: dict[str, int]
v = {"a": 1, "b": 0}
v["c"] = 2
print(v)
```

Un caso más complicado sería un diccionario con un tipo enumerado como índice:

```
from enum import Enum
tipoColor = Enum('tipoColor',
    ['rojo', 'verde', 'azul', 'amarillo'])
v: dict[tipoColor, int]
v = {tipoColor.rojo: 1, tipoColor.verde: 4, tipoColor.azul: 5}
v[tipoColor.amarillo] = 3
```

El vector `v` se representa como un diccionario cuyas claves son colores y cuyos valores son números enteros. Posteriormente creamos un vector con 3 valores (las posiciones de los colores en el arco iris) y a continuación añadimos un nuevo valor para el valor amarillo, con el objetivo de ilustrar dos posibles maneras de añadir valores.

La siguiente cuestión a considerar es cómo iterar por todas las posiciones de un vector. En el caso de los índices numéricos comenzando en 0, hemos visto que podíamos recorrer con un bucle `for` todos los valores entre 0 y el tamaño del vector (no incluido), accesible usando la función `len`. En el caso de índices numéricos que no comiencen en 0, podríamos adaptarlo de manera sencilla:

```
for i in range(posInicial, posInicial + len(v)):
    print(v[i], end='')
```

siendo `posInicial` el índice de la primera posición del vector. Para el ejemplo anterior de un array de caracteres, sería 1.

Si, como en este caso, simplemente estamos interesados en acceder a los contenidos del vector, se puede iterar sobre ellos:

```
for dato in v:
    print(dato, end='')
```

Veamos ahora el caso de índices no numéricos. En el caso de que los índices sean tipos enumerados podemos hacer un bucle que itere sobre todos los posibles valores del tipo enumerado. En el ejemplo anterior de un vector de colores podríamos hacer:

```
for i in tipoColor:
    print(v[i], end='␣')
```

En el caso de un vector cuyas claves son caracteres podríamos hacer:

```
for codigo in range(ord(primerIndice), ord(finalIndice) + 1):
    i = chr(codigo)
    print(v[i], end='␣')
```

donde `primerIndice` es el índice de la primera posición (en nuestro ejemplo, 'a') y `finalIndice` es el índice de la última posición (en nuestro ejemplo, 'c'). Obsérvese que

a `ord(finalIndice)` se le suma uno, ya que el segundo argumento de la función `range` es el máximo valor, pero no incluido, y en este caso queremos incluir también a la última letra. Recordemos que `ord` devuelve el código numérico asociado a un carácter, y que `chr` devuelve el carácter asociado a dicho código numérico.

9.7. Mínimo de un vector

En las siguientes secciones vamos a considerar algunos problemas típicos de manejo de vectores: cálculo del mínimo, ordenación, búsqueda de un elemento, inserción de un elemento y borrado de un elemento.

Un algoritmo sencillo para calcular el mínimo valor de un vector consiste en comenzar asumiendo que el mínimo es el elemento situado en la primera casilla. A partir de ahí, se recorren las demás posiciones del vector, comparando el valor de cada casilla con el valor que hasta ahora era el mínimo. Si en algún momento se encuentra un valor inferior al que hasta ese momento era el mínimo, se actualiza el valor del mínimo con dicho valor. Veamos una implementación en Pascal:

```
TYPE
   vector = ARRAY[1..20] OF real;

FUNCTION minimo(v : vector; tam : integer): real;
VAR
   i : integer;
   min : real;
BEGIN
   min := v[1];
   FOR i := 2 TO tam DO
      IF v[i] < min THEN
         min := v[i];
   minimo := min;
END;
```

Los parámetros que recibe la función son el vector `t` y el tamaño real del vector `tam`. El algoritmo supone que el vector no está vacío, o sea, `tam > 0`, y que `tam <= high(v)`, por lo que, antes de llamar a esta función, habría que asegurarse de que se cumplen dichas condiciones. Además, por simplicidad, también se supone que la primera posición del vector (`low(v)`) es la `1`

Una variante habitual es buscar el máximo de un vector. Por ejemplo, dado un vector de notas, podemos dividir todas las notas entre la nota máxima para normalizar las calificaciones. Para calcular el máximo de un vector, bastaría con reemplazar `v[i] < min` con `v[i] > min`. Adicionalmente, la variable `min` debería renombrarse a `max`.

Veamos ahora un equivalente en Python del cálculo del mínimo:

```
vector = list[float]

def minimo(v: vector) -> float:
    i: int
    m: float
    m = v[0]
    for i in range(len(v)):
        if v[i] < m:
            m = v[i]
    return m
```

Otra variante interesante, bastante usada en la práctica, consiste en calcular la posición en el vector del valor mínimo:

```
FUNCTION posMinimo(v : vector; tam : integer): real;
VAR
   i, posMin : integer;
BEGIN
   posMin := 1;
   FOR i := 2 TO tam DO
      IF v[i] < v[posMin] THEN
         posMin := i;
   posMinimo := posMin;
END;
```

Obsérvese que si existan varias apariciones del valor mínimo, se devolvería la menor posición de todas ellas. Otra variante interesante sería devolver todas las posiciones en las que aparece el valor mínimo.

Veamos cómo se adaptaría el código anterior a Python:

```
def posMinimo(v: vector) -> int:
    i: int
    posMin: int
    posMin = 0
    for i in range(len(v)):
        if v[i] < v[posMin]:
            posMin = i
    return posMin
```

9.8. Ordenación de un vector

Ordenar un vector tiene muchas aplicaciones. Evidentemente, facilita las búsquedas: digan lo que digan los adolescentes, es más fácil encontrar las cosas en un lugar ordenado (sea un dormitorio o un vector) que en otro desordenado.

A modo de ejemplo vamos a considerar ordenaciones de vectores de números reales en orden ascendente. Modificarlos para considerar orden descendente es trivial y considerar vectores de otro tipo de dato tampoco presenta ninguna dificultad siempre que exista una manera de comparar si un dato es mayor que otro.

Entre los múltiples algoritmos de ordenación vamos a considerar un par de ellos que no son especialmente eficientes pero sí son intuitivos y sencillos de entender pero también de recordar y de implementar si hace falta en un momento dado. Para ver esto de una manera divertida, con bailarines danzando al ritmo de los algoritmos de ordenación, recomiendo los vídeos del canal de Youtube de Algorythimcs [4][1].

En Python sería tan sencillo como invocar a la función `sort` de las listas. Dado un `vector` `v`, podríamos simplemente escribir:

```
v.sort()
```

Aún así, nos parece interesante ver con cierto detalle cómo podrían implementarse, también en Python, algunos algoritmos de ordenación: selección (en inglés, *selection sort*) y burbuja (en inglés, *bubble sort*).

Ampliación 17. *La función* `sort` *de Python implementa un algoritmo más eficiente,* timsort*, que combina algoritmos de mezcla (*merge sort*) e inserción (*insertion sort*).*

Comencemos por el algoritmo de selección. La idea es calcular en el paso j-ésimo la posición del mínimo del subvector que queda por ordenar e intercambiarlo por el elemento en la j-ésima posición del vector. De esta manera, al final del paso j, las j primeras posiciones quedarán ordenadas. Un posible código en Pascal sería el siguiente:

```
1  PROCEDURE ordenarSeleccion(VAR v: vector; tam : integer);
2  VAR
3      aux : real;
4      i, j, posMin : integer;
5  BEGIN
6      FOR j := 1 TO tam - 1 DO
7      BEGIN
8          posMin := j;
9          FOR i := j + 1 TO tam DO
10             IF v[i] < v[posMin] THEN
11                 posMin := i;
12         aux := v[j];
13         v[j] := v[posMin];
14         v[posMin] := aux
15     END
16 END;
```

[1] http://www.youtube.com/@AlgoRythmics/videos

Efectivamente, en las líneas 8–11 se calcula la posición del mínimo del subvector que hay entre las posiciones j y tam del vector a ordenar, y se almacena en la variable `posMin`. A continuación, las líneas 12–14 intercambian el contenido de las casillas `posMin` y `j`. En principio, este proceso debería repetirse tantas veces como casillas tenga el vector. Sin embargo, tal y como vemos en la línea 6, es suficiente con repetirlo $tam - 1$ veces, porque si están correctamente ordenadas las primeras $tam - 1$ casillas, la última también debe haber quedado automáticamente ordenada.

En Python, podría traducirse del siguiente modo:

```
def ordenarSeleccion(v: vector) -> None:
    i: int
    posMin: int
    for i in range(len(v)):
        posMin = i
        for j in range(i+1, len(v)):
            if v[j] < v[posMin]:
                posMin = j
        v[i], v[posMin] = v[posMin], v[i]
```

Veamos ahora la ordenación de burbuja. La idea ahora es comparar cada elemento con el que tiene ubicado justamente al lado e intercambiarlos si procede. Este procedimiento debe realizarse hasta que el vector quede ordenado. Una manera eficiente de comprobarlo consiste en detectar que se haya realizado una iteración completa sin que haya habido ningún intercambio sobre el vector.

Una posible implementación de este algoritmo en Pascal sería:

```
PROCEDURE ordenarBurbuja(VAR v : vector; tam : integer);
VAR
    aux : real;
    i : integer;
    intercambio : boolean;
BEGIN
    REPEAT
        intercambio := false;
        FOR i := 1 TO tam - 1 DO
            IF v[i] > v[i+1] THEN
            BEGIN
                aux := v[i];
                v[i] := v[i+1];
                v[i+1] := aux;
                intercambio := true
            END;
    UNTIL intercambio = false
END;
```

Un elemento con un valor muy grande se intercambiaría con el siguiente elemento, se volvería a intercambiar con el siguiente elemento y así muchas veces. A alguien le debió parecer que esos valores muy grandes se asemejaban a burbujas de aire ascendiendo hasta la superficie del agua, y de ahí el nombre del algoritmo[2].

Observemos que la variable booleana `intercambio` se inicializa a falso en cada iteración del bucle `REPEAT` y que cada vez que hay un intercambio se cambia su valor a verdadero. Por lo tanto, para que la condición del bucle `REPEAT` sea verdadera, debe haber una iteración completa sin que haya ningún intercambio.

Una versión alternativa en Python (usando un bucle `while` porque en Python no hay un equivalente del `REPEAT`) sería la siguiente:

```
1  def ordenarBurbuja(v: vector) -> None:
2      i: int
3      intercambio: bool
4      intercambio = True
5      while intercambio:
6          intercambio = False
7          for i in range(len(v) - 1):
8              if v[i] > v[i+1]:
9                  v[i], v[i+1] = v[i+1], v[i]
10                 intercambio = True
```

Observemos que la variable booleana `intercambio` se inicializa a falso dentro del bucle `while` (línea 6), por lo que, para que se repita el bucle, debe haber algún intercambio (líneas 9–10) que cambie el valor de la variable a verdadero. Para que el bucle se ejecute al menos una vez, `intercambio` se inicializa a verdadero justo antes de comenzar el bucle (línea 4).

9.9. Búsqueda en un vector

Básicamente, el problema consiste en encontrar la posición en la que aparece un elemento dado en un vector. Esto admite múltiples variantes, como qué hacer si el elemento aparece más de una vez y qué hacer si no aparece ninguna. Si aparece más de una vez podríamos devolver la posición de la primera ocurrencia, la última, alguna cualquiera o todas ellas: optaremos por devolver la que tengo un menor índice. Por otro lado, si no aparece ninguna, optaremos por devolver una posición inválida para el vector (como -1), pero hay otras opciones como generar una excepción.

No obstante, al igual que en la sección anterior, vamos a considerar dos algoritmos: la búsqueda secuencial general y la búsqueda binaria para un caso particular. Veremos

[2]Por cosas como esa, desaconsejamos absolutamente el consumo de drogas.

cómo codificarlos tanto en Pascal como en Python. Consideraremos vectores de reales con números enteros como índices (comenzando desde el 1 en Pascal y desde 0 en Python), aunque es inmediatamente generalizable a otros vectores.

En Pascal, el algoritmo de búsqueda secuencial sería:

```
FUNCTION busquedaSecuencial(v: vector; tam: integer;
                            elBuscado: real) : integer;
VAR
    i, pos  : integer;
    exito : boolean;
BEGIN
    i := 1;
    pos := -1;
    exito := false;
    WHILE (i <= tam) AND (NOT exito) DO
    BEGIN
        IF v[i] = elBuscado THEN
        BEGIN
            pos := i;
            exito := true;
        END
        ELSE
            i := i + 1
    END;
    busquedaBinaria := pos
END;
```

Es decir, se considera una doble condición de parada: haber llegado al final del bucle o haber encontrado el elemento buscado. Inicialmente, se usa una variable `exito` inicializada a falso que indica que no se ha encontrado el elemento todavía. En coherencia con ello, la variable `pos`, que almacena el resultado a devolver, se inicializa a -1. A partir de ahí, se recorre cada casilla del vector: si en la casilla i-ésima del vector se encuentra el valor buscado, se modifica el valor de `pos` a `i` y se actualiza `exito` a verdadero (para que salga del bucle).

Debemos advertir que Python directamente proporciona la función `index` que devuelve la posición de un elemento en una lista. Sin embargo, en caso de que no exista se genera una excepción que debe gestionarse. En el siguiente código veremos cómo gestionar una excepción: se intenta ejecutar el bloque del `try` (línea 4) y, en caso de producirse una excepción, se intenta capturar en el bloque del `except` (línea 5). Si la excepción que se genera durante la ejecución coincide con la especificada tras el `except` (en este caso, la excepción `ValueError`) se ejecuta el bloque de código del `except` (línea 6). Es decir, si `index` devuelve una posición, `busqueda` la devuelve. En otro caso, se produce una excepción `ValueError` y `busqueda` devuelve -1.

```
1 def busqueda(v: vector, elBuscado: float) -> int:
2     pos: int
3     try:
4         pos = v.index(elBuscado)
5     except ValueError:
6         pos = -1
7     return pos
```

Una traducción directa a Python del algoritmo de búsqueda secuencial anterior es:

```
def busquedaSecuencial(v: vector, elBuscado: float) -> int:
    i: int
    pos: int
    exito: bool
    i = 1
    pos = -1
    exito = False
    while (i <= len(v)) and (not exito):
        if v[i] == elBuscado:
            pos = i
            exito = True
        else:
            i = i + 1
    return pos
```

En el caso especial de un vector ordenado, puede usarse una búsqueda binaria:

```
FUNCTION busquedaBinaria(v: vector; tam: integer;
                         elBuscado: real) : integer;
VAR
    izq, dch, centro, pos : integer;
BEGIN
    izq := 1;
    dch := tam;
    centro := (izq + dch) DIV 2;
    WHILE (izq <= dch) AND (v[centro] <> elBuscado) DO
    BEGIN
        IF v[centro] > elBuscado THEN
            dch := centro - 1
        ELSE
            izq := centro + 1;
        centro := (izq + dch) DIV 2;
    END;
    IF v[centro] = elBuscado THEN
        pos := centro
    ELSE
        pos := -1;
    busquedaBinaria := pos
END;
```

Como vemos, este algoritmo es más sofisticado y eficiente. Se usan dos variables `izq` y `dch` para indicar los extremos del subvector donde se busca en cada momento. Inicialmente, se busca en el vector completo, entre las posiciones 0 y $tam - 1$. Dependiendo del valor del elemento en la posición central del subvector (indicada por la variable `centro`), seguiremos buscando en el subvector comprendido entre 0 y $centro - 1$, seguiremos buscando en el subvector ubicado entre $centro + 1$ y $tam - 1$, o finalizaremos la búsqueda si en la posición `centro` se halla el elemento buscado.

Es algo parecido a lo que se hacía al buscar una palabra en un diccionario en papel: abrir por una posición aleatoria o aproximada y comparar la palabra buscada con la palabra encontrada: si nuestra palabra es alfabéticamente menor, buscamos en las páginas a la izquierda, y si nuestra palabra es alfabéticamente superior, buscamos en las páginas a la derecha.

Observemos también que hay dos motivos por los que podría salir del bucle `WHILE`: `v[centro]` podría ser igual al `elBuscado` y el valor de `izq` podría ser mayor que el de `dch`. Por tanto, al salir del bucle debemos comprobar por qué motivo ha salido: si `v[centro]` es igual al `elBuscado`, hemos encontrado el elemento que buscábamos y devolvemos la posición `centro`; en otro caso, no lo hemos encontrado tras recorrer todo el vector y devolvemos -1.

Una posible codificación en Python sería la siguiente:

```
def busquedaBinaria(v: vector, elBuscado: float) -> int:
    izq: int
    dch: int
    centro: int
    izq = 0
    dch = len(v) - 1
    centro = (izq + dch) // 2
    while (izq <= dch) and (v[centro] != elBuscado):
        if v[centro] > elBuscado:
            dch = centro - 1
        else:
            izq = centro + 1
        centro = (izq + dch) // 2
    if v[centro] == elBuscado:
        return centro
    else:
        return -1
```

Recordemos que en Python `//` calcula el cociente de la división como un número entero, por lo que es un índice válido para el vector.

9.10. Inserción en un vector

Dado un vector $[4, 15, 16, 23, 42]$, si insertamos en el valor 8 en la posición 2, obtenemos el vector $[4, 8, 15, 16, 23, 42]$ (suponiendo que las posiciones se numeran a partir de la uno).

Para insertar un elemento en un vector, debemos especificar la posición y el valor del nuevo elemento. Además, en el caso de Pascal, como los arrays son estáticos, debemos tener en cuenta tanto el tamaño real como el tamaño máximo (y solo podemos insertar si el tamaño real usado es inferior al máximo). Un posible código sería el siguiente:

```
1  PROCEDURE insertar(VAR v: vector; pos: integer; nuevo: real;
2                     VAR tam: integer; tamMax: integer);
3  VAR
4      i : integer;
5  BEGIN
6      IF (tam < tamMax) AND (pos >= 1) AND (pos <= tam + 1) THEN
7      BEGIN
8          FOR i := tam DOWNTO pos DO
9              v[i + 1] := v[i];
10         v[pos] := nuevo;
11         tam := tam + 1
12     END;
13 END;
```

Lo primero que se hace, en la línea 6, es comprobar que efectivamente el vector no está lleno, es decir, que el tamaño real (`tam`) es inferior al tamaño máximo (`tamMax`). A continuación se comprueba que la posición es correcta, estando entre `1` (para insertar a la izquierda del primer elemento) y `tamMax` +1 (para insertar a la derecha del último elemento).

Seguidamente, las líneas 8–9 copian algunos elementos del vector a la derecha para dejar sitio al nuevo elemento, concretamente los elementos entre las posiciones `pos` y `tam`. Es importante que esta operación se haga comenzando desde el final del vector, para no sobrescribir valores del vector al copiar a la derecha. Una vez que se ha hecho hueco al nuevo elemento, se inserta en la nueva posición (línea 10). Por último, se actualiza el tamaño real del vector, que ahora contiene un elemento más (línea 11).

Ejemplo 28. *Supongamos que queremos insertar el valor* 8 *en la posición* 2 *del vector* `[4, 15, 16, 23, 42]`*. El bucle se repite desde la posición* `tam` (5) *hasta la* `pos` (2)*, en orden decreciente:*

- *Para* $i = 5$*, el vector queda* `[4, 15, 16, 23, 42, 42]`
- *Para* $i = 4$*,* `[4, 15, 16, 23, 23, 42]`

- *Para* $i = 3$, `[4, 15, 16, 16, 23, 42]`
- *Para* $i = 2$, `[4, 15, 15, 16, 23, 42]`

Finalmente, tras copiar el nuevo valor en la línea 10*, el resultado es* `[4, 8, 15, 16, 23, 42]`.

Recordando que en Free Pascal están disponibles las funciones `low(v)` y `high(v)` para devolver el menor y el mayor posible valor, respectivamente, de los índices de un vector `high(v)`, podríamos hacer un procedimiento `insertar` que no recibiera como parámetro el tamaño máximo del vector. Además, no necesitaría suponer que los índices comienzan a numerarse a partir del uno:

```
1  PROCEDURE insertar(VAR v: vector; pos: integer; nuevo: real;
2                     VAR tam: integer);
3  VAR
4      i : integer;
5  BEGIN
6      IF (tam < high(v) - low(v) + 1) AND (pos >= low(v))
7              AND (pos <= low(v) + tam) THEN
8      BEGIN
9          FOR i := tam DOWNTO pos DO
10             v[i + 1] := v[i];
11         v[pos] := nuevo;
12         tam := tam + 1
13     END;
14 END;
```

Por un lado, para comprobar si el vector está vacío se comprueba si el tamaño real `tam` es estrictamente menor a la diferencia entre `high(v)` y `low(v)` más uno. Por ejemplo, si los índices van desde el 1 hasta el 10, `tam` debe ser estrictamente menor a $10 - 1 + 1 = 10$.

Por otro lado, se evalúa si la posición `pos` es válida comprobando que esté en el intervalo comprendido entre el menor índice posible `low(v)` (incluido) y la posición que está `tam` posiciones a lad derecha (no incluida).

Una implementación similar a la de Pascal sería:

```
1  def insertar(v: vector, pos: int, nuevo: float) -> None:
2      i: int
3      v.append(0)
4      if (pos >= 0) and (pos <= len(v)):
5          for i in range(len(v) - 2, pos - 1, -1):
6              v[i+1] = v[i]
7          v[pos] = nuevo
```

La diferencia principal reside en la gestión del tamaño de la lista. En Python no es necesario almacenar los tamaño real y máximo ni incrementar el tamaño real, pero sí es necesario ampliar la lista para que tenga espacio para un elemento más. Esto se hace en la línea 3, donde `append` añade al final de la lista un nuevo elemento. Podría ser el elemento que hasta ese momento estaba en la última posición, pero para contemplar la posibilidad de que la lista de entrada esté vacía, se añade un valor cualquiera (ya que va a ser sobrescrito en seguida). Dado que en Python las posiciones se numeran a partir del cero, las posiciones correctas para la inserción son las que están entre `0` (para insertar a la izquierda del primer elemento) y la longitud de la lista (para insertar a la derecha del último elemento). Esto se comprueba en la línea 4.

Las líneas 5–6 copian copian algunos elementos del vector a la derecha para dejar sitio al nuevo elemento, al igual que en Pascal. Cabe aclarar que `range(len(v) - 2, pos - 1, -1)` devuelve los valores en el intervalo $[len(v) - 2, pos - 1)$ en orden decreciente (debido al tercer parámetro -1). Puede parecer extraño que a la longitud del vector se le resten dos; uno porque acabamos de insertar un nuevo elemento y otro porque las posiciones comienzan en cero. Finalmente, se copia el nuevo elemento en la posición `pos` (línea 7).

Ejemplo 29. *Para aclarar el funcionamiento de* `range` *en este caso, veamos un ejemplo concreto, el mismo que en Pascal: insertar el valor* 8 *en la posición* 1 *(pues se numeran desde cero) de la lista* `[4, 15, 16, 23, 42]`*. Tras crear un nuevo elemento en la línea* 3*, la lista queda* `[4, 15, 16, 23, 42, 0]`*. Ahora,* $len(v) - 2 = 4$*, y* $pos - 1 = 0$*. Por tanto,* `range` *devuelve los valores* 4, 3, 2 *y* 1 *(pues el primer argumento de* `range` *está incluido pero el segundo no):*

- *Para* $i = 4$*, la lista queda* `[4, 15, 16, 23, 42, 42]`
- *Para* $i = 3$*,* `[4, 15, 16, 23, 23, 42]`
- *Para* $i = 2$*,* `[4, 15, 16, 16, 23, 42]`
- *Para* $i = 1$*,* `[4, 15, 15, 16, 23, 42]`

Finalmente, tras copiar el nuevo valor en la línea 7*, el resultado es la lista* `[4, 8, 15, 16, 23, 42]`*.*

Por último, mencionemos que en Python la inserción de un nuevo elemento en una lista sería tan sencilla como

```
v.insert(pos, nuevo)
```

siendo evidentemente `pos` la posición del nuevo elemento y `nuevo` su valor.

9.11. Borrado en un vector

Dado un vector $[4, 8, 15, 16, 23, 42]$, si borramos en valor de la posición 2 (suponiendo que se numeran a partir de la uno), obtenemos el vector $[4, 15, 16, 23, 42]$.

Para borrar un elemento de vector, solo debemos especificar la posición. También podría especificarse solo el valor del nuevo elemento, pero en este caso habría qué aclarar qué sucede si el elemento aparece otras veces. En el caso de Pascal, también hay que tener en cuenta el tamaños real.

En Pascal, el elemento no se va a borrar realmente, en el sentido de que físicamente el vector ocupará el mismo tamaño en memoria, pero se va a decrementar el tamaño real del vector, para que no se accedan a las posiciones que exceden el tamaño real:

```
1  PROCEDURE borrar(VAR v: vector; pos: integer;
2                   VAR tam: integer);
3  VAR
4      i : integer;
5  BEGIN
6      IF (tam > 0) AND (pos >= 1) AND (pos <= tam) THEN
7      BEGIN
8          FOR i := pos TO tam - 1 DO
9              v[i] := v[i+1];
10         tam := tam - 1
11     END;
12 END;
```

La línea 6 comprueba que el vector no está vacío y que la posición es correcta, estando entre la primera y la correspondiente al tamaño real del vector. Las líneas 8–9 desplazan algunos elementos a la izquierda, sobrescribiendo el elemento que queremos borrar. Concretamente, los elementos entre las posiciones `pos` y `tam - 1` se reemplazan por los elementos situados a su derecha. Finalmente, se actualiza el tamaño real del vector, que ahora contiene un elemento menos (línea 10).

En Python, sería tan sencillo como usar la función `pop`, si bien puede generar una excepción `ValueError` si en la lista `v` no existe la posición `pos`:

```
def borrar(v: vector, pos: int) -> None:
    if (pos >= 0) and (pos < len(v)):
        v.pop(pos)
```

9.12. Cadenas de caracteres

Una cadena de caracteres es un array de caracteres con algunos operadores especiales definidos. Hasta ahora, hemos usado las cadenas de caracteres como constantes

para escribir mensajes por pantalla. Sin embargo, dada una cadena de caracteres, podemos usar los corchetes para acceder a una posición concreta de la cadena, al igual que sucede con los vectores.

Veamos a modo de ejemplo el cálculo de la letra correspondiente a un determinado número de Documento Nacional de Identidad, teniendo en cuenta que las posiciones comienzan a numerarse a partir del uno:

```
PROGRAM letraDNI;
CONST
    letras = 'TRWAGMYFPDXBNJZSQVHLCKE';
VAR
    dni, indice : integer;
BEGIN
    write('Introducir un DNI: ');
    readln(dni);
    { Suponiendo (dni >= 0) AND (dni <= 99999999) }
    indice := (dni MOD 23) + 1
    writeln('Letra: ', letras[indice]);
END.
```

En Python podríamos hacer algo similar, teniendo en cuenta que ahora las posiciones comienzan a numerarse a partir del cero:

```
letras: str
dni: int
indice: int
letras = "TRWAGMYFPDXBNJZSQVHLCKE"
print("Introducir un DNI: ", end='')
dni = int(input())
indice = (dni % 23)
print("Letra:", letras[indice])
```

En Pascal hay 2 mecanismos básicos para manejar cadenas de caracteres, `ARRAY OF char` y `STRING`. Es Python, tenemos `str`. Veamos cada uno de estos casos.

ARRAY OF char. Es una cadena de caracteres de tamaño fijo, al igual que los arrays convencionales. A veces se usa un `PACKED ARRAY OF char`, que permite minimizar el espacio de almacenamiento (si las cadenas tienen muchos caracteres, quizá conviene optimizar el espacio). La definición del tipo es similar a la de los arrays:

```
TYPE
    tipoCadena = PACKED ARRAY[1..15] OF char;
```

Para dar valor a una cadena, podemos leerlo de teclado o asignar un valor (una constante o una expresión que devuelva un valor de ese tipo):

```
VAR
    cadena : tipoCadena;
BEGIN
    cadena := 'Hola␣mundo';
END.
```

Por defecto, en todas las posiciones de un `ARRAY OF char` hay un carácter especial, `chr(0)`. Si a una variable de ese tipo se le asigna una cadena de menor tamaño que el tamaño máximo, las casillas no utilizadas mantienen ese carácter especial, `chr(0)`. Por ejemplo, para la cadena del ejemplo anterior:

'H'	'o'	'l'	'a'	' '	'M'	'u'	'n'	'd'	'o'	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

¡Atención! 16. *Es buena práctica de programación evitar que las cadenas de caracteres contengan el carácter* `chr(0)` *como un carácter válido, porque diferentes compiladores lo interpretan de diferente manera, como veremos en la Ampliación 18.*

Al contrario de lo que sucede con otros tipos de arrays, `read` y `readln` sí que saben leer cadenas de caracteres. Además, como hemos visto en ejemplos anteriores (desde nuestro primer programa en Pascal), `write` y `writeln` también saben escribir por pantalla cadenas de caracteres. `write` y `writeln` gestionan el tamaño real, de modo que las posiciones no usadas no se muestran.

Ampliación 18. *A la hora de calcular el tamaño real por* `write` *y* `writeln`*, hay diferencias según la versión del compilador. En algunas versiones (como en la versión 3.0.4), el tamaño real se corresponde con la posición del primer* `chr(0)` *que aparezca, comenzando a buscar desde la primera posición, menos uno. En otras versiones (como en la 3.2.2), se comienza por el final de la cadena y se busca la primera posición distinta de* `chr(0)`*. Ambos algoritmos coinciden si desde el primer* `chr(0)` *hasta el final de la cadena todo son caracteres* `chr(0)`*.*

La función `length` devuelve el tamaño máximo de un `ARRAY OF char`:

```
writeln(length(cadena));
```

Ampliación 19. *El funcionamiento de* `length` *también depende de la versión de Free Pascal. En versiones anteriores, como en la 3.0.4,* `length` *solo estaba permitido si el* `ARRAY OF CHAR` *tenía al menos* 256 *caracteres.*

Para calcular el tamaño real de un `ARRAY OF char`, se puede recorrer la cadena de caracteres desde la primera posición buscando la posición anterior al primer `chr(0)`:

```
i := 0;
WHILE (i < tam) AND (cadena[i+1] <> chr(0)) DO
    i := i + 1;
longitud := i;
```

Ampliación 20. *Suponemos que después del primer* `chr(0)` *no hay caracteres válidos; de lo contrario se obtendría distinto resultado si comenzamos desde el final de la cadena buscando la posición anterior al último* `chr(0)`*.*

Para las cadenas de caracteres están permitidos los operadores de igualdad y desigualdad. `=` compara si dos cadenas de caracteres tienen el mismo valor (la misma longitud, y los mismos caracteres en todas las posiciones entre uno y la longitud de la cadena). `<>` compara si la condición anterior no se cumple.

También están permitidos los operadores de comparación. Por ejemplo, `<` compara si una cadena es menor que otra de la siguiente manera: si el primer elemento de la primera cadena es menor que el primer elemento de la segunda, devuelve `true`. Si son iguales, entonces compara los segundos elementos de la primera cadena y de la segunda cadena, y así sucesivamente. Recordemos que para comparar dos caracteres se tiene en cuenta el número entero que los codifica en la tabla ASCII. Por ejemplo, `'abcd' < 'abd'` devuelve `true` porque el tercer carácter (`'c'`) de la primera es menor que el tercer carácter (`'d'`) de la segunda; y los caracteres primero y segundo son iguales. Cuidado: `'Ab < AC'` devuelve `false` porque `'b'` es mayor que `'C'` en la tabla ASCII, aunque aparezca antes en el alfabeto.

El operador `+` se usa para indicar la concatenación de cadenas, es decir, para anexar una al final de la otra. Por ejemplo, `'Hola' + 'Mundo'` devolvería `'HolaMundo'`.

¡Atención! 17. *Tengamos cuidado si el primer operando de la concatenación es una variable de tipo* `ARRAY OF CHAR`*: la segunda cadena se anexará a partir de la posición siguiente al tamaño máximo de la primera cadena, no a partir del tamaño real.*

Por último, recordemos que, al igual con cualquier otro vector, se permite la asignación, siempre que la variable y el nuevo valor asignado tengan exactamente el mismo tipo.

STRING. También es una cadena de caracteres con algunas operaciones adicionales definidas. Al crear un `STRING`, se puede indicar un tamaño máximo o no (en este caso, se considera el tamaño máximo por defecto, 255 caracteres).

```
TYPE
    tipoCadena = STRING; { 255 caracteres }
    tipoCadena100 = STRING[100];  { 100 caracteres }
```

Ampliación 21. *En realidad, existen internamente dos tipos de* `STRING` *en Free Pascal:* `ShortString` *(de longitud máxima 255) y* `AnsiString`*, sin limitación. Una opción del compilador permite que* `STRING` *se interprete como* `AnsiString`*. Para no depender del compilador, es una buena práctica de programación ser conservadores y suponer que la longitud máxima de un* `STRING` *es 255.*

Están permitidas todas las operaciones adicionales que hemos mencionado anteriormente para los `ARRAY OF CHAR`: longitud, lectura de teclado, escritura por pantalla, operadores relacionales y concatenación.

¡Atención! 18. *La concatenación de una variable de tipo* `STRING` *funciona de manera algo diferente: la segunda cadena se anexará a partir de la posición siguiente al tamaño real de la primera cadena.*

Ampliación 22. *Según la documentación de Free Pascal, para un* `STRING` *de hasta* 255 *caracteres, existe una manera adicional de acceder a la longitud: acceder a la posición cero de la cadena. Es decir, dada la variable* `cadena` *de tipo* `STRING`*,* `cadena[0]` *devuelve un entero con su posición. Sin embargo, no funciona en las últimas versiones de Free Pascal, como 3.0.4 ó 3.2.2.*

Además de lo anterior, existen varias funciones disponibles para las cadenas de tipo `STRING`, entre las cuales destacaremos las siguientes:

- Búsqueda de una subcadena dentro de una cadena. La función

  ```
  pos(subcadena, cadena)
  ```

  devuelve la primera posición de la subcadena, o 0 si no se encuentra

- Extracción de una subcadena a partir de una cadena. La función

  ```
  subcadena = copy(cadena, pos, numCaracteres)
  ```

  devuelve el `STRING` entre las posiciones `pos` (incluida) y `pos + numCaracteres` (no incluida) de `cadena`.

- Inserción de una subcadena en la posición `pos` de una cadena, a través del procedimiento

  ```
  insert(cadena, subcadena, pos)
  ```

- Borrado de una subcadena dada una cadena. El procedimiento

```
delete(cadena, pos, numCaracteres)
```

borra la subcadena entre las posiciones `pos` (incluida) y `pos + numCaracteres` (no incluida).

- Transformación de un número real (o entero) a cadena, por medio del procedimiento:

```
str(numeroReal, cadena)
```

- Transformación de una cadena (que codifique un valor numérico como cadena de caracteres) en un número real, por medio del procedimiento:

```
val(cadena, numeroReal, error)
```

Después de llamar al procedimiento, la variable `error` indicará 0 si no se produjeron errores en la conversión, o bien un número positivo que indique la primera posición de la cadena donde se produjo el error.

str. Una vez vistas las cadenas de caracteres de Pascal, veamos el caso de Python. Ya explicamos que el tipo `str` [8] permite representar cadenas de caracteres y que, de hecho, no existe el tipo carácter como tal, por lo que para manejar caracteres usábamos cadenas de longitud 1.

En este apartado resumiremos brevemente las diferencias con las cadenas de caracteres de Pascal:

- Para conocer la longitud, disponemos de la función `len`.
- Para acceder a una posición concreta (para inspeccionar su valor) se usan los corchetes, pero las posiciones comienzan a numerarse en la posición cero.
- Las cadenas de caracteres son inmutables, por lo que no pueden modificar su valor: se debe crear un objeto nuevo. Para insertar una subcadena en la posición `pos`, sin borrar el valor de dicha posición, podríamos hacer:

```
cadena = cadena[:pos] + subcadena + cadena[pos+1:]
```

En caso de querer modificar el valor de la posición `i` con un carácter `c`, podríamos hacer:

```
if pos < len(srt):
    cadena = cadena[:pos] + c + cadena[pos+1:]
elif pos = len(srt):
    cadena = cadena[:pos] + c
```

- Para borrar la subcadena situada entre las posiciones `posIni` (incluida) y `posFinal` (no incluida), podríamos hacer:

```
cadena = cadena[:posIni] + subcadena + cadena[posFinal:]
```

- Los literales de las cadenas de caracteres (es decir, sus posibles valores) se pueden representar tanto con comillas simples como con comillas dobles.

- La búsqueda de una subcadena dentro de una cadena se usa con la función `find`, que tiene como parámetros opcionales las posiciones inicial (incluida) y final (excluida) donde buscar, devolviendo -1 en caso de no encontrarla:

```
pos = cadena.find('texto', 0, len(pos))
```

- Como ya hemos estudiado para leer de teclado, las funciones `int` y `float` permiten transformar una cadena de caracteres en un entero y un real, respectivamente.

- Para realizar el camino inverso, la función `str` recibe un número y devuelve una cadena de caracteres.

9.13. Ejercicios

Ejercicio 16. *Uno de los muchos indicadores propuestos para cuantificar la calidad del trabajo realizado por los investigadores es el índice H. El índice H de un investigador es x si posee al menos x publicaciones que han recibido al menos x citas cada una. La idea subyacente es que a una mayor cantidad de citas recibidas por sus publicaciones científicas corresponde una mayor calidad.*

La información relativa a un investigador puede representarse utilizando un array de enteros, de forma que el índice del array indica el orden que ocupa cada publicación en el total de su producción científica y el valor de la posición i del array indica el número de citas recibidas por el artículo i-ésimo.

Por ejemplo, si un investigador ha publicado 3 artículos tales que el primero ha recibido 4 citas, el segundo ha recibido 5 citas y el tercero no ha recibido ninguna, podríamos utilizar el siguiente array:

4	*5*	*0*
1	*2*	*3*

En este caso, el índice H del array sería 2*, porque existen* 2 *artículos (el primero y el segundo) que han recibido al menos* 2 *citas cada uno.*

Escriba una función que reciba como entrada un array de enteros (y, en Pascal, el tamaño real vector) y que devuelva como salida el índice H.

Ejercicio 17. *Dos átomos de un elemento químico son isótopos si tienen el mismo número atómico pero diferente número másico (es decir, igual número de protones pero diferente de neutrones). Por ejemplo, el hidrógeno tiene* 3 *isótopos estables: protio (*0 *neutrones), deuterio (*1 *neutrón) y tritio (*2 *neutrones). No todos los elementos químicos tienen el mismo número de isótopos y existen elementos sin ningún isótopo estable.*

Un elemento químico puede representarse junto a sus isótopos estables (con el tipo de dato `isotopos`*) almacenando su número atómico y una lista de tantos números másicos como isótopos estables existan. Si no existe ninguno, esta lista estaría vacía. Los números atómicos y másicos son de tipo entero (además, siempre toman valores estrictamente positivos) y se sabe que el elemento con más isótopos estables (el estaño) tiene* 10*. Una posible representación para una tabla periódica de isótopos estables de los elementos requeriría un vector de elementos de tipo* `isotopos` *con tamaño* `NUM_ELEMENTOS` *(en la actualidad,* 118*).*

Supongamos las siguientes definiciones en Pascal (o su equivalente en Python):

```
CONST
    MAX_ISOTOPOS = 10;
    NUM_ELEMENTOS = 118;
TYPE
    isotopos = RECORD
        numAtomico : integer;
        numIsotopos : 0..MAX_ISOTOPOS;
        numMasico : ARRAY[1..MAX_ISOTOPOS] OF integer;
    END;
    tablaIsotopos = ARRAY[1..NUM_ELEMENTOS] OF isotopos;
```

Implemente un procedimiento que reciba una variable de tipo `tablaIsotopos` *como parámetro y escriba por pantalla cuál es el elemento que tiene* 2 *isótopos estables con una mayor diferencia de números másicos entre ellos, así como el valor de dicha diferencia. Para calcular la diferencia de números másicos de los isótopos de un elemento dado, se calculará el máximo de los números másicos de sus isótopos y se le restará el mínimo de los números másicos de sus isótopos. En caso de existir varios isótopos con la misma diferencia, se considerará el que tenga menor número atómico.*

Por ejemplo, dada la siguiente representación del hidrógeno y sus 3 *isótopos, el mínimo número másico es* 1*, el máximo es* 3 *y la diferencia es* $3 - 1 = 2$*:*

1	3	1	2	3	···	?
numAtomico	*numIsotopos*	*numMasico[1]*	*numMasico[2]*	*numMasico[3]*	...	*numMasico[10]*

Ejercicio 18. *El estado de oxidación de un elemento químico es un número entero que representa el número de electrones que un átomo recibe o aporta para la formación de un compuesto. Si los electrones se reciben, el signo del estado de oxidación es negativo; si se aportan, el signo es positivo. Para nosotros no tendrá interés un estado de oxidación igual a* 0*, pues corresponde a átomos aislados.*

Un elemento puede tener diferentes estados de oxidación. Por ejemplo, el hidrógeno tiene 2 *(*-1 *y* 1*), el oxígeno tiene* 4 *(*-2*,* -1*,* 1 *y* 2*) y el bario tiene* 1 *(*2*). Otros, como el helio, no tienen ninguno.*

El mayor número posible de estados de oxidación de un elemento no puede ser superior a 12*, pues el mayor estado de oxidación conocido es* 8 *(por ejemplo, para el rutenio) y el menor es* -4 *(por ejemplo, para el carbono), lo que supone* 8 *estados de oxidación positivos y* 4 *negativos.*

Una posible representación para una tabla periódica de elementos podría contener, para cada elemento, su número atómico y una lista (no necesariamente ordenada) de sus posibles estados de oxidación. Podemos suponer que la lista no incluye un estado de oxidación igual a 0 *(pues este estado corresponde a átomos aislados) y que se tienen las siguientes definiciones en Pascal (o su equivalente en Python):*

```
CONST
    MAX_ESTADOS = 12;
    NUM_ELEMENTOS = 118;
TYPE
    estadoOxidacion = -4..8;
    elemento = RECORD
        numAtomico : integer;
        numEstados : 1.. MAX_ESTADOS;
        estados : ARRAY[1.. MAX_ESTADOS] OF estadoOxidacion;
    END;
    tablaPeriodica = ARRAY[1.. NUM_ELEMENTOS] OF elemento;
```

Escriba una función que reciba 2 *parámetros de tipo* `elemento` *y compruebe si existe o no algún estado de oxidación del primer elemento y algún estado de oxidación del segundo elemento tal que la suma de ambos sea igual a* 0*. Esta condición es necesaria para que haya un enlace entre un átomo de cada elemento.*

Por ejemplo, si la función recibe los datos del hidrógeno y el oxígeno, la respuesta es positiva, pues los estados -1 *(del hidrógeno) y* 1 *(del oxígeno) suman* 0*. Sin embargo, si recibe el hidrógeno y el bario, la respuesta es negativa.*

Ejercicio 19. *Basándose en el Ejercicio 18, escriba una función que reciba* 2 *parámetros de tipo* `elemento` *y devuelva un valor que indique si representan o no a la misma entidad. Para que esos* 2 *parámetros representen la misma entidad, deben tener el mis-*

mo número atómico y los mismos estados de oxidación (sin que uno de ellos tengo un estado que no tenga el otro).

Ejercicio 20. *En 1997, el campeón del mundo de ajedrez, Garry Kasparov, fue derrotado por el supercomputador Deep Blue, desarrollado por la empresa IBM (por cierto, en el equipo que desarrolló Deep Blue estaba el español Miguel Illescas, informático y varias veces campeón de España de ajedrez). Se considera uno de los hitos de la Inteligencia Artificial, pues por primera vez las máquinas conseguían vencernos en un juego de inteligencia donse siempre se pensó que los humanos serían superiores por la imposibilidad de calcular todas las posibles jugadas en una partida de ajedrez.*

Consideremos el tipo `tipoTableroAjedrez` *definido en la Sección 9.4 (o su equivalente en Python). Escriba un subprograma que cree un tablero para representar el estado de la partida de la Figura 9.1, que se produjo justo antes de que Garry Kasparov abandonara en la última partida del enfrentamiento.*

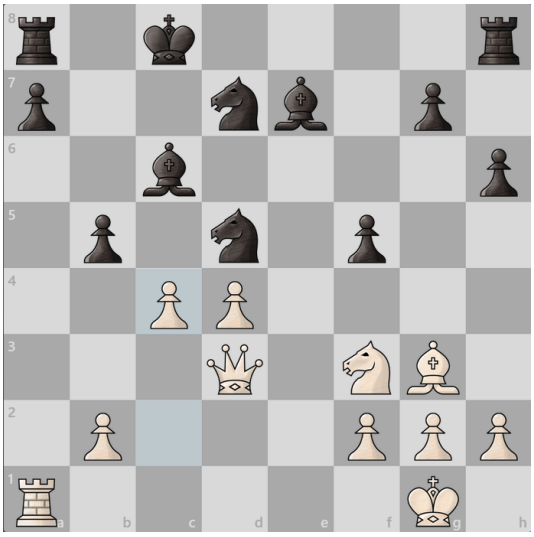


Figura 9.1: Estado de un partida de ajedrez.

Ejercicio 21. *Escriba un subprograma que reciba un* `tipoTableroAjedrez` *definido en la Sección 9.4 (o su equivalente en Python) y compruebe si el estado de la partida es correcto de acuerdo con los siguientes criterios:*

- *Cada banda tiene exactamente un rey.*
- *Ningún bando tiene más de ocho peones.*
- *Ningún bando tiene más de un alfil del mismo color*[3].

[3]Estrictamente, un peón podría coronar y cambiarse por un alfil, pero descartamos ese caso tan improbable

Ejercicio 22. *Consideremos el código genético de las proteínas (Ejercicio 9), determinado por cinco bases nitrogenadas diferentes. La adenina (A), la guanina (G) y la citosina (C) se encuentran tanto en el ADN como en el ARN, la timina (T) sólo en el ADN y el uracilo (U) sólo en el ARN.*

El código genético puede representarse mediante una tabla de codones, definida según los siguientes tipos de datos de Pascal (o su equivalente en Python):

```
TYPE
    aminoacido = (AcidoAspartico, AcidoGlutamico, Alanina,
                  Arginina, Asparagina, Cisteina, Especial,
                  Fenilalanina, Glicina, Glutamina,
                  Histidina, Isoleucina, Leucina, Lisina,
                  Metionina, Prolina, Serina, Tirosina,
                  Treonina, Triptofano, Valina);
    base = (T, A, C, G, U);
    tablaCodones = ARRAY[base, base, base] OF aminoacido;
```

Por ejemplo, en la posición `[A,A,A]` *de la tabla ADN está la* `Lisina`*. En la definición de* `aminoacido` *el valor* `Especial` *no corresponde a ningún aminoácido sino que permite codificar los significados especiales antes mencionados de inicio o final de secuencia.*

Construya un procedimiento que reciba como parámetros un aminoácido, un booleano y una tabla de codones y escriba por pantalla un listado de todos los codones (uno por línea) que codifican dicho aminoácido. Si el valor booleano es verdadero, se considerará el caso del ADN. De lo contrario, se considerará el ARN.

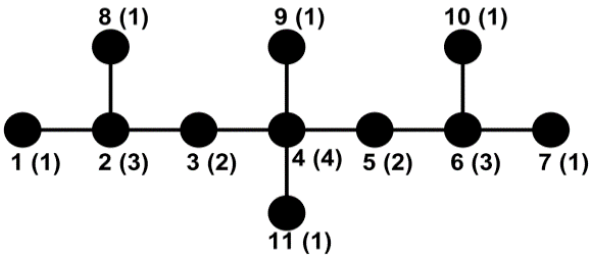


Figura 9.2: Grafo molecular del 2,4,4,6-tetrametilheptano, donde para cada vértice se muestra entre paréntesis el número de vértices adyacentes

Ejercicio 23. *El esqueleto de átomos de carbono de una molécula orgánica se puede representar mediante un grafo molecular $G = (V, E)$. El conjunto de vértices V son los átomos de carbono de la molécula, y el conjunto de arcos E son enlaces carbono-carbono entre pares de vértices, como ilustra la Figura 9.2.*

	1	2	3	4	5	6	7	8	9	10	11
1	0	**1**	0	0	0	0	0	0	0	0	0
2	**1**	0	**1**	0	0	0	0	**1**	0	0	0
3	0	**1**	0	**1**	0	0	0	0	0	0	0
4	0	0	**1**	0	**1**	0	0	0	**1**	0	**1**
5	0	0	0	**1**	0	**1**	0	0	0	0	0
6	0	0	0	0	**1**	0	**1**	0	0	**1**	0
7	0	0	0	0	0	**1**	0	0	0	0	0
8	0	**1**	0	0	0	0	0	0	0	0	0
9	0	0	0	**1**	0	0	0	0	0	0	0
10	0	0	0	0	0	**1**	0	0	0	0	0
11	0	0	0	**1**	0	0	0	0	0	0	0

Tabla 9.1: Matriz de incidencia del 2,4,4,6-tetrametilheptano.

Un grafo molecular puede representarse mediante una matriz de incidencia M, con una fila y una columna por cada vértice. Dados dos vértices cualesquiera α y β, si son adyacentes (existe un arco que los conecta), entonces $M[\alpha, \beta] = M[\beta, \alpha]1$, sino $M[\alpha, \beta] = M[\beta, \alpha] = 0$. La Tabla 9.1 muestra la matriz de incidencia asociada a la molécula de la Figura 9.2.

Escriba una función que reciba como parámetro de entrada una matriz de incidencia `M` (que representa un grafo molecular) y compruebe si la matriz es correcta o no. Para considerarse correcta, `M` debe verificar todas las siguientes condiciones:

- *`M` debe ser cuadrada;*
- *`M` debe ser simétrica;*
- *Todos los elementos de la diagonal principal de `M` deben ser nulos; y*
- *Ningún vértice de `M` está desconectado(es decir, para todo vértice existe algún arco que lo conecta con otro vértice).*

Suponga las siguientes definiciones de tipos en Pascal (o su equivalente en Python):

```
TYPE
    matrizIncidencia = RECORD
        numFilas, numColumnas : 1..100;
        matriz : ARRAY [1..100, 1..100] OF 0..1;
    END;
```

Ejercicio 24. *Para facilitar la búsqueda de información en bases de datos y en la web, se han propuesto diferentes notaciones para identificar sustancias químicas codificando su información molecular usando solo caracteres ASCII. Un ejemplo es la*

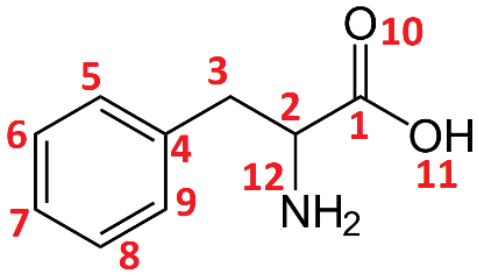


Figura 9.3: Molécula de fenilalanina.

notación ROSDAL, que se describe a continuación. A cada átomo se le asigna arbitrariamente un número entero excepto a los átomos de hidrógeno, que no se muestran. Los átomos de carbono se muestran indicando solo su número asociado. Para el resto de elementos químicos, se muestra además del número su símbolo químico. Los enlaces simples, dobles y triples entre parejas de átomos se representan mediante los caracteres `-`, `=` *y* `#`*, respectivamente. Las distintas ramas de la molécula se separan mediante comas. La notación ROSDAL es inequívoca (se puede identificar la molécula sin ninguna ambigüedad) pero no es única para una molécula dada.*

Por ejemplo, para la molécula de la fenilalanina (Figura 9.3), dos representaciones distintas podrían ser "1-2-3-4=5-6=7-8=9-4,1=10O,1-11O,2-12N" y "4-9=8-7=6-5=4-3-2-12N,2-1=10O,1-11O".

Escriba un programa que lea de teclado una molécula en notación ROSDAL y escriba por pantalla el número total de átomos de elementos distintos al hidrógeno en la molécula. Se supone que la entrada no contiene errores y que no hay más de 15 *átomos distintos al hidrógeno.*

Ejercicio 25. *Considere el Ejercicio 24 y construya un programa que lea de teclado una molécula en notación ROSDAL y escriba por pantalla el número de átomos de carbono que contiene dicha molécula. Se supone que la entrada no contiene errores y que no hay más de* 100 *átomos en una molécula que no sean hidrógeno. Por ejemplo, para cualquiera de las dos cadenas anteriores para la fenilalanina, la respuesta sería* 9 *átomos de carbono.*

Ejercicio 26. *Una manera habitual de representar moléculas químicas es usar una matriz B-E (*Bond-Electron matrix*), donde cada celdilla c_{ij} representa el número de enlaces que hay entre los átomos i y j ($i \neq j$) de la molécula, y cada casilla de la diagonal c_{ii} indica el número de electrones de valencia libres. Por ejemplo, dada la molécula del etanal (Figura 9.4, donde a cada átomo se le asigna un valor numérico del* 1 *al* 7*, al lado del símbolo químico), la Tabla 9.2 muestra la matriz B-E equivalente.*

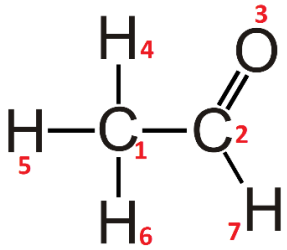


Figura 9.4: Molécula de etanal.

	1	2	3	4	5	6	7
1	0	1	0	1	1	1	0
2	1	0	2	0	0	0	1
3	0	2	4	0	0	0	0
4	1	0	0	0	0	0	0
5	1	0	0	0	0	0	0
6	1	0	0	0	0	0	0
7	0	1	0	0	0	0	0

Tabla 9.2: Matriz B-E asociada al etanal.

*Uno de los problemas de la matriz B-E es que su tamaño crece de manera cuadrática con el número de átomos de la molécula, por lo que es ineficiente para moléculas grandes. Como variante, se puede utilizar una lista de enlaces (*bond list*), que almacena para cada pareja de átomos i, j el número de enlaces entre ellos utilizando un array en vez de una matriz, como se observa en la Tabla 9.3.*

Diseñe el tipo de dato `listaEnlaces` *que permita representar listas de enlaces con un número en enlaces no superior a* `MAX_ENLACES` *y construya un procedimiento que reciba una matriz B-E (de tipo* `matrizBE`*) y calcule la* `listaEnlaces` *equivalente. Se supone que existen las siguientes definiciones (o sus equivalentes en Python):*

```
CONST
    MAX_ATOMOS = 10;
    MAX_ENLACES = 10;
TYPE
    matrizBE = RECORD
        numAtomos : 1..MAX_ATOMOS;
        matriz : ARRAY[1..MAX_ATOMOS, 1..MAX_ATOMOS] OF
                 0..MAX_ENLACES;
    END;
    enlace = ...
    listaEnlaces = ...
```

Ejercicio 27. *Considere el Ejercicio 26 en el sentido contrario: construya un procedimiento que reciba una* `listaEnlaces` *y calcule una matriz* `matrizBE` *equivalente.*

Fila	Átomo1	Átomo2	Enlaces
1	1	2	1
2	1	4	1
3	1	5	1
4	1	6	1
5	2	3	2
6	2	7	1

Tabla 9.3: Lista de enlaces asociada al etanal.

Ejercicio 28. *Una de las aplicaciones de la Inteligencia Artificial en Ingeniería Química está en el descubrimiento de fármacos (*drug discovery*) haciendo una predicción de la estructura 3D de la proteína diana (*target protein*) para un tratamiento exitoso de la enfermedad. Al predecir la estructura de la proteína en 3D, el diseño del fármaco será conforme al entorno químico de incidencia en la proteína diana, lo que ayuda a predecir su efecto en la diana junto con consideraciones de seguridad antes de su síntesis o producción.*

La estructura 3D de una proteína en un ordenador se puede representar mediante vóxeles. Un vóxel es la unidad mínima de volumen en el espacio 3D con un color propio. Por tanto, una imagen digital 3D de la estructura de una proteína se puede representar como una matriz tridimensional de vóxeles, donde cada vóxel codifica el color de una parte (un cubo) de la estructura. Para el problema a resolver no es relevante la forma de asignar color a un vóxel a partir de la estructura de la proteína.

*Para representar el color en un vóxel, se usará el modelo RGB, donde el color se especifica mediante tres números en el intervalo [0, 255]: la componente roja (*Red*), la componente verde (*Green*) y la componente azul (*Blue*). Por ejemplo, el color {0, 0, 0} corresponde al negro, el color {255, 0, 0} al rojo y el color {255, 0, 255} al morado.*

Para representar estructuras 3D de proteínas dentro de una matriz tridimensional de $100 \times 100 \times 100$ *vóxeles se disponen de las siguientes definiciones y estructuras de datos en Pascal (o sus equivalentes en Python):*

```
CONST
    MAX_VOXELES = 100;
TYPE
    color = ARRAY[1..3] OF 0..255;
    estructura = RECORD
        numVoxeles : 0..MAX_VOXELES;
        matriz3D : ARRAY[1..MAX_VOXELES, 1..MAX_VOXELES,
                         1..MAX_VOXELES] OF color;
    END;
```

Construya una función que reciba dos parámetros de tipo `estructura`*, correspondientes a una estructura real sintetizada en un laboratorio y a otra obtenida por ordenador. La función debe calcular el grado de similitud entre ambas estructuras, medido como el porcentaje de vóxeles iguales en ambas estructuras sobre el total de vóxeles de la estructura real que NO son negros.*

Capítulo 10

Ficheros

Hasta ahora, todos los programas que hemos desarrollado se basaban en el uso de la memoria principal, que es no permanente, para almacenar datos e instrucciones. Como consecuencia, los programas requerían introducir la entrada del programa tecleándola desde el teclado y simplemente mostraban la salida del programa por pantalla. Esto resulta aceptable en algunos casos pero, en general, puede ser problemático. Por un lado, no queremos que la salida del programa se pierda al apagar el ordenador: frecuentemente deseamos que el resultado se pueda almacenar en el ordenador, posiblemente para ser utilizado por otros programas. Además, cuando la entrada de datos tiene un gran tamaño (por ejemplo, si es necesario suministrar a un programa la tabla periódica) es bastante tedioso tener que teclearla en cada ejecución del programa: sería preferible poder usar datos que ya estuvieran almacenada en el ordenador.

Como solución al problema descrito, en este capítulo veremos cómo hacer que nuestros programas utilicen la memoria secundaria (que es permanente) como entrada y/o salida de datos, en vez de usar la memoria principal.

En la memoria secundaria, la información se almacena en ficheros. Un fichero o archivo es un conjunto de información homogénea, tratada como una unidad de almacenamiento y organizada de forma estructurada para la recuperación de cualquier dato individual. Es fácil pensar en ejemplos de ficheros y de formatos de ficheros. Al escribir estas líneas, se están almacenando en un fichero de texto cuyo formato viene dato por el lenguaje de composición de textos LaTeX[1]. Si estás leyendo en formato digital (espero que legalmente y con respeto a los derechos de autor), probablemente tendrás ante un fichero en formato pdf.

El sistema operativo permite realizar numerosas operaciones con los ficheros, como por ejemplo borrarlos, renombrarlos o copiarlos a otra carpeta a otra unidad. En nuestros programas, también podremos manejar ficheros. Típicamente, los lenguajes de

[1] Es absolutamente recomendable aprender a usar LaTeXde cara a futuros documentos de gran tamaño y complejidad, especialmente el Trabajo Fin de Grado [6].

programación proporcionan algunos mecanismos para poder usar algunos de los servicios del sistema operativo para el manejo de ficheros. Como mínimo, podremos realizar las dos operaciones básicas: leer y escribir ficheros. Es importante, por tanto, tener en cuenta que nuestros programas permitirán manipular los ficheros utilizando el sistema operativo como intermediario.

En particular, a la hora de indicar el nombre y la ruta de un fichero, debemos seguir las convenciones de cada sistema operativo en particular. Lar uta puede ser relativa o absoluta. Una ruta relativa es incompleta, indicando solamente la parte de la ruta que se debe añadir a la ubicación actual. Por ejemplo, si nos referimos al fichero "nombre.dat", el sistema supondrá que el fichero está en la carpeta actual (es decir, en el directorio donde esté ubicado el ejecutable de nuestro programa). En una ruta absoluta, se indica la ruta completa, por lo que la ubicación del fichero es independiente de la ubicación del ejecutable del programa. Por ejemplo, en un sistema operativo Windows, una ruta absoluta podría ser "C:\Mis Documentos\nombre.dat", mientras que en un sistema operativo macOS una ruta absoluta podría ser "/Users/nombreUsuario/nombre.dat".

En resumen, las variables permanecen en memoria principal, tienen un tamaño máximo fijo, requieren que se les asigne un valor inicial y su valor se pierde al finalizar el programa. En cambio, los ficheros permanecen en memoria secundaria, no tienen tamaño máximo (el tamaño máximo lo determina la capacidad del dispositivo de almacenamiento secundario), no necesitan que desde el programa se les de un valor inicial y su valor no se pierde con el fin de la ejecución del programa.

10.1. Formatos de ficheros

El formato de un fichero es el modo en el que internamente se representa la información. Por ejemplo, una memoria de prácticas de una asignatura desarrollada con Microsoft Word suele tener formato .docx. Sin embargo, a la hora de entregarla, podemos entregarla en formato .pdf. Son dos formatos diferentes, cada uno con sus ventajas e inconvenientes: es preferible escribir la memoria en .docx pero al entregarla en .pdf nos aseguraremos de que no haya cambios sobre la manera en que se visualiza el fichero en ordenadores diferentes al nuestro. Típicamente, los nombres de los ficheros son de la forma nombre1.nombre2, si bien esto no es más que una convención y no es en absoluto obligatorio. La cadena de texto que hay tras el punto se denomina *extensión* y suele hacer referencia al formato del fichero, aunque no es obligatorio. Siguiendo con el ejemplo anterior, la memoria de prácticas en formato .docx podría estar en un fichero prácticas.docx y la memoria final en otro fichero prácticas.pdf. La extensión, por ejemplo, indica al sistema operativo con qué aplicación abrir el fichero cuando hace-

mos doble click sobre él. Evidentemente, para cambiar el formato de un fichero, no es suficiente con renombrarlo cambiando su extensión: es necesario leer el fichero (en el formato original) y escribirlo (en el formato deseado). Por suerte, muchas aplicaciones permiten convertir de .docx a .pdf.

Podemos diferenciar dos grandes familias de ficheros, los ficheros binarios y los ficheros de texto. Los ficheros binarios, como su nombre indica, representan la información directamente en código binario. Aunque hay muchos tipos de formatos binarios, nos centraremos en los formatos compatibles con los lenguajes de programación Pascal y Python. Si nos planteamos cómo se puede ver o crear un fichero binario desde un lenguaje de programación, la respuesta es que, exclusivamente, usando un programa en dicho lenguaje de programación que permita leer o escribir ficheros. Otras aplicaciones (como editores de texto, procesadores de textos, etc.) no permiten, en general, trabajar con esos tipos de ficheros.

Los ficheros de texto, en cambio, representan la información mediante un conjunto de líneas, cada una de las cuales incluye una secuencia (posiblemente vacía) de caracteres. Aunque, internamente, esos caracteres se representan en binario, se utiliza una representación intermedia (por ejemplo, el código ASCII), de modo que ahora, se puede ver o crear un fichero de texto desde un lenguaje de programación, pero también desde cualquier editor de texto, como el Bloc de notas de Windows. Un fichero de texto es parecido a un fichero binario de caracteres, pero donde los caracteres se estructuran en líneas (aunque los saltos de línea también se codifican como caracteres).

Como ventajas de los ficheros de texto, tenemos una mayor portabilidad (son conocidos por más aplicaciones), una mayor legibilidad por humanos (pues entendemos el texto mejor que el código binario) y una mayor flexibilidad para representar datos con diferente tipo en diferentes líneas (como veremos, en Pascal generalmente todos los datos de los fichero binarios son del mismo tipo). Como desventajas, requieren un mayor tiempo de procesamiento, al tener que convertir de un formato de representación de caracteres (como el código ASCII) a formato binario (que es lo que entiende el ordenador), un mayor espacio en disco (por ejemplo, en Pascal un real ocupa 4 Bytes en un fichero binario, mientras que en un fichero de texto requiere 1 Byte por cada dígito) y un desarrollo de los programas algo más complicado (al tener que manejar las líneas de los ficheros).

En este libro vamos a considerar estos tipos de ficheros: los ficheros binarios, en el formato usado por nuestro lenguaje de programación, y los ficheros de texto. Por cierto, los ficheros donde escribimos el código fuente de los programas son ficheros de texto y los ficheros con el programa ejecutable son binarios (aunque en un formato diferente a los ficheros binarios que usaremos desde nuestros programas).

10.2. Modos de acceso a los ficheros

Hay dos tipos fundamentales de acceso a los datos de un fichero: secuencial y aleatorio. En el acceso secuencial, en un momento dado se puede acceder únicamente a un único dato, el que esté ubicado en una posición concreta. Esto sucede también en otros dispositivos de música o vídeo, como una cinta de cassette o una cinta VHS: en cada momento solo se puede acceder al dato que está junto al cabezal. En cambio, el acceso aleatorio permite que en un momento dado se pueda acceder a cualquier dato, independientemente de su posición. Esto sucede en un disco de vinilo o en una memoria USB. En este curso, solo consideraremos el acceso secuencial.

Para analizar cómo manejar los ficheros secuenciales, puede resultar recordar cómo funcionan una cinta de cassette o una cinta VHS, siendo conscientes de que son tecnologías algo obsoletas y que muchos de los jóvenes estudiantes no las habrán visto en funcionamiento.

En una cinta de cassette, la información está codificada en una larga tira de plástico, que se encuentra enrollada. La cinta tiene dos bobinas con capacidad de giro, lo que permiten avanzar o retroceder la cinta. Al leer una cinta, se procede a hacerla girar lentamente, de modo que en cada momento se accede a un único dato (lo que está exactamente frente al cabezal de lectura) y de modo que avanza la cinta, por lo que una vez leído un dato, el dato disponible para la lectura será el siguiente anterior. Si al principio la cinta estaba rebobinada, la cinta comenzará desde el principio. En otro caso, la lectura comenzará por el dato que quede frente al cabezal. La lectura termina cuando se llegue al final de la cinta o cuando el usuario la detenga.

Los ficheros tienen asociado un puntero, que indica la posición actual del fichero que se está procesando. Los ficheros secuenciales también se procesan dato a dato, comenzando desde el primero (si el fichero está "rebobinado") o desde la posición indicada por el puntero del fichero. Una vez que se lee un dato, el puntero avanza, para que el siguiente dato disponible fuera el siguiente. Típicamente, se usa un bucle de lectura mientras que no se cumpla una condición de parada, que puede ser alcanzar el final del fichero o hasta que se cumpla una condición que dependa del problema (por ejemplo, podemos querer leer los 10 primeros datos de un fichero).

En las siguientes secciones veremos, por este orden, cómo realizar la lectura de ficheros binarios, la escritura de ficheros binarios, la lectura de ficheros de texto y la escritura de ficheros de texto.

10.3. Lectura de ficheros binarios

Con un poco más de nivel de detalle sobre lo manera de gestionar ficheros secuenciales que mencionamos anteriormente, la lectura de un fichero es un proceso que se compone de los siguientes pasos:

- Abrir el fichero, especificando el nombre y la ruta del fichero que se quiere leer e indicando que se quiere trabajar en modo lectura. Esto causará que se asocie el fichero a una variable administrativa local y que se rebobine el fichero.
- Leer un dato del fichero, almacenándolo en una variable.
- Repetir el paso anterior hasta que se cumpla una cierta condición de parada.
- Cerrar el fichero, disociándolo de la variable administrativa y permitiendo que el fichero esté libre para que otras aplicaciones puedan moverlo, modificarlo, borrarlo, etc. Seguramente alguna vez hemos intentando borrar un fichero y el sistema operativo nos lo ha impedido porque el fichero estaba siendo usado por alguna aplicación: cerrar el fichero evita ese tipo de problemas.

Comencemos nuestra exposición mostrando cómo leer ficheros binarios en Pascal. Para ello, vamos a crear un programa que permita leer un fichero binario dato a dato. A modo de ejemplo, consideraremos un fichero de enteros, lo leeremos completamente y escribiremos por pantalla cada dato leído, para poder verificar que el programa funciona y realmente estamos accediendo a todos los datos del fichero. En Pascal, el código sería:

```
1  PROGRAM leerFichero;
2  TYPE
3     tipoDato = integer;
4     tipoFichero = FILE OF tipoDato;
5  VAR
6     dato : tipoDato;
7     f : tipoFichero;
8     nombre : STRING;
9  BEGIN
10    write('Escriba␣nombre␣del␣fichero:␣');
11    readln(nombre);
12    assign(f, nombre);
13    reset(f);
14    WHILE (NOT eof(f)) DO
15    BEGIN
16       read(f, dato);
17       write(dato, '␣');  { Hacer algo con el dato }
18    END;
19    close(f)
20 END.
```

Expliquemos brevemente el código:

- En las líneas 2–3 se define el tipo de los datos (`integer`) y el tipo del fichero, que permitirá manejar ficheros de enteros. Al igual que en este lenguaje todas las variables tienen un tipo asignado, casi todos los ficheros binarios también lo tienen. En Pascal, manejaremos normalmente ficheros binarios de tipo T, que serán sucesiones de datos, todos ellos de tipo T. Podemos tener ficheros de enteros, ficheros de reales o, en general, ficheros de cualquier tipo que se pueda definir en Pascal (excepto ficheros de ficheros).

- En las líneas 6–8, definimos las variables que permitirán almacenar un dato leído del fichero, su nombre (y ruta) y una variable administrativa del fichero `f`. Para manejar ficheros (que existen más allá de nuestros programas) desde nuestro programas, usamos unas variables administrativas de fichero que hacen de enlace entre ambos. A veces se llama ficheros lógicas a estas variables administrativas de los ficheros físicos. Otras veces se las llama simplemente ficheros, lo que es desaconsejado porque puede inducir a errores.

- En las líneas 10–11, se solicita al usuario que escriba por teclado el nombre del fichero que se desea leer. Este proceso no sería necesario en aquellos casos en los cuales ya se conozca el nombre del fichero.

- En la línea 12, el procedimiento `assign` vincula una variable administrativa de fichero (`f`) con un fichero físico, cuyo nombre se indica mediante una cadena de caracteres (`nombre`), que debe contener la ruta (relativa o absoluta) del fichero.

- En la línea 13, el procedimiento `reset` indica que se va a trabajar en modo lectura con el fichero `f` y se posiciona al principio del archivo. En aquellos casos en los que el fichero no esté al principio (si se está en medio de una lectura anterior) la llamada a `reset` forzaría el "rebobinado" del fichero. En Pascal, un fichero puede manejarse en modo lectura o en modo escritura, pero no en ambos modos a la vez.

- En la línea 14, la función `eof` (del inglés *end of file*) devuelve un valor booleano que indica si se ha llegado al final del fichero, es decir, devuelve `true` si el puntero del fichero apunta al final del mismo, o `false` en otro caso contrario.

- En la línea 16, el procedimiento `read` lee un dato del fichero `f` y lo almacena en la variable `dato`. `read`, que ya conocemos para leer el valor de una o varias variables de teclado, también permite que el valor de las variables se lea de un fichero,

siempre que se pase como primer parámetro en la llamada al procedimiento una variable administrativa de fichero.

Sin embargo, hay una diferencia notable. Tal y como vimos, Pascal sabe cómo leer de teclado los tipos de dato entero, real, carácter y cadena de caracteres, pero no sabe cómo leer los tipos de datos definidos por el programador (como registros o arrays). En ese caso, la solución es leer los tipos de datos estructurados por partes, leyendo de manera independiente cada campo del registro o cada casilla del array. En cambio, cuando se lee de un fichero binario, Pascal sí sabe cómo leer los tipos de datos definidos por el programador, por lo que se puede leer directamente de teclado un tipo de dato estructurado. De hecho, si tenemos un fichero binario de datos de tipo T, es obligatorio que `read` lee, en cada llamada al procedimiento, un dato de tipo T, siendo incorrecto leer de un fichero de registros solo un campo del registro o leer de un fichero de arrays una casilla del array.

- La línea 17 simplemente ilustra un uso posible del dato leído: escribirlo por pantalla. Evidentemente, esta línea se sustituirá en nuestros programas por un procesamiento diferente (y más complejo).

- Una vez que se sale del bucle de la línea 14 porque se ha llegado al final del fichero, la línea 19 cierra el fichero, disociándolo del fichero físico al que estuviera asociado y permitiendo que esté disponible para otras aplicaciones.

Recomendamos encarecidamente al lector que pruebe este programa en su ordenador. Evidentemente, previamente debe haberse creado un fichero de enteros; en la Sección 10.4 veremos cómo hacerlo.

¡Atención! 19. *Al usar* ***`read`*** *(pero también* ***`eof`****) es importante tener en cuenta que el funcionamiento es diferente si en la llamada se pasa como parámetro un fichero o no: en el primer caso se tiene en cuenta el fichero, pero en el segundo caso se tiene en caso la entrada estándar (el teclado).*

En caso de un tipo de dato definido por el programador, la única observación destacable es que el tipo de los datos del fichero debe definirse antes de definir el tipo del fichero. Por ejemplo:

```
TYPE
    tipoEstudiante = RECORD
        nombre : STRING;
        id : integer;
        nota : real;
    END;
    tipoFicheroEstudiantes = FILE OF tipoEstudiante;
```

La lectura de un dato del fichero sería exactamente igual,

```
read(f, dato);
```

siendo en este caso `dato` de tipo `tipoEstudiante`.

Otra consideración importante es que, si conocemos en el momento de desarrollo del programa el nombre del fichero, podemos asignarlo directamente sin preguntárselo al usuario. Veamos un par de ejemplos para sustituir las líneas 10–12 del código anterior, para una ruta relativa y para una ruta absoluta:

```
assign(f, 'fichero.dat');
assign(f, 'C:\Ejemplos␣Pascal\fichero.dat');
```

Por otro lado, debemos hacer constar que el tipo de datos `FILE OF` es especial y tiene algunas particularidades:

- No puede existir un fichero de ficheros.
- No puede ser el resultado de una función.
- En caso de que se pase como parámetro a una función o procedimiento, debe pasarse como variable de entrada/salida. El motivo es que sus efectos secundarios (como avanzar el puntero de un fichero después de una lectura) siempre son visibles desde el módulo que invocó al subprograma.
- No se utilizan operadores aritméticos, relacionales, lógicos o de asignación, sino únicamente las funciones y procedimientos específicamente definidas para los ficheros, como `assign`, `reset`, `read`, `eof` y `close`. Al estudiar la escritura de ficheros y los ficheros de texto, completaremos esta lista con `rewrite`, `write` y `eoln`. Existen algunos subprogramas adicionales que no veremos en el curso, pero que pueden encontrarse en la documentación de Free Pascal [2].

Es trivial modificar el bucle para que finalice la lectura de un fichero si se cumple alguna condición (por ejemplo, tras leer un número negativo). No obstante, cabe subrayar que debemos comprobar que se cumplan las dos condiciones para permanecer en el fichero: no haber llegado al final del fichero y no haber encontrado un valor negativo, es decir:

```
dato := 0;
WHILE (NOT eof(f)) AND (dato >= 0) DO
BEGIN
   read(f, dato);
   write(dato, '␣');  { El negativo también se escribe }
END;
```

¡Atención! 20. *Son errores frecuentes:*

- *Asociar los ficheros pero no llamar a* `reset` *para iniciar la lectura*
- *Abrir un fichero inexistente o indicando una ruta incorrecta*
- *Leer tras llegar al final del fichero*
- *Usar* `eof` *sin argumento (el compilador no lo detecta como error)*
- *Leer sin abrir el fichero*
- *Leer habiendo abierto el fichero en modo escritura*
- *Leer un tipo de dato que no se corresponde con el tipo de dato del fichero*
- *Leer de fichero un campo de un registro*

A riesgo de resultar cansinos, vamos a volver a incidir en un aspecto: la diferencia de comportamiento de `read` en Pascal cuando se lee de fichero y cuando se lee de fichero binario. En la siguiente código, las líneas 15–16 son correctas, mientras que las líneas 14 y 17 son incorrectas. En efecto, desde un fichero binario se puede leer un registro completo (línea 15), pero no un campo del registro (línea 17). Desde teclado, en cambio, se puede leer un campo del registro, por ser de tipo entero (línea 16), pero no se puede leer un registro completo (línea 14).

```
1  PROGRAM leerFicheroYteclado;
2  TYPE
3     tipoRegistro = RECORD
4       a : integer;
5       b : real;
6     END;
7     tipoFicheroRegistros = FILE OF tipoRegistro;
8  VAR
9     f : tipoFicheroRegistros;
10    unRegistro : tipoRegistro;
11 BEGIN
12    assign(f, 'ficheroPrueba.dat');
13    reset(f);
14    read(unRegistro);       { Incorrecto }
15    read(f, unRegistro);    { Correcto }
16    read(unRegistro.a);     { Correcto }
17    read(f, unRegistro.a); { Incorrecto }
18    close(f)
19 END.
```

Ampliación 23. *En algunos dialectos de Pascal, incluyendo Pascal estándar pero no Free Pascal, existe un operador especial, la variable búfer de un fichero, que devuelve el siguiente dato del fichero sin mover el puntero del fichero. Al poder saber el dato que nos vamos a encontrar antes de que, aparentemente, se lea, se simplifica el desarrollo de ciertos programas. La sintaxis es* `f^`*, siendo* `f` *una variable administrativa de fichero.*

Cambiemos de tercio y veamos cómo resolver el problema anterior (leer un fichero de enteros y escribir su contenido por pantalla) en Python. Si bien en este lenguaje el abanico de posibilidades a la hora de escribir en fichero es mucho más amplio, hemos optado para el uso de `Struct`, por considerar que es la alternativa más parecida a Pascal. El código Pascal anterior podría traducirse a Python del siguiente modo:

```
1  import io
2  import struct
3  dato: int
4  eof: bool
5  f: io.BufferedReader
6  formatoEntero: struct
7  misBytes: bytes
8  nombre: str
9  print("Escriba␣nombre␣del␣fichero:␣", end='')
10 nombre = input()
11 f = open(nombre, 'rb')
12 formatoEntero = struct.Struct('i')
13 numBytes = struct.calcsize('i')
14 eof = False
15 while not eof:
16     misBytes = f.read(numBytes)
17     if not misBytes:
18         eof = True
19     else:
20         r = formatoEntero.unpack_from(misBytes)
21         dato = r[0]
22         print(dato, end='␣')  # Hacer algo con el dato
23 f.close()
```

Pasemos ahora a explicar el código:

- En las líneas 1–2 se añaden los paquetes `io` y `struct`, necesarios para usar `BufferedReader` y `Struct`, respectivamente.

- En las líneas 3–8 se declaran las variables que usaremos en nuestro programa. Al igual que antes, tendremos variables para almacenar un dato leído del fichero (`dato`), su nombre y ruta (`nombre`) y una variable administrativa del fichero

(`f`). Ahora, además, tendremos variables para almacenar el formato de los datos fichero (`formatoEntero`), el dato leído representado como una lista de Bytes (`misBytes`) y una variable booleana para controlar cuándo salir del bucle (`seguir`).

Mientras que en Pascal existe el concepto de fichero binario de tipo T, en Python esto no es así, todos los ficheros binarios son colecciones de bytes, por lo que un fichero binario puede tener datos de distinto tipo (representados como bytes). De hecho, ahora tenemos el tipo `BufferedReader`, que permite leer ficheros binarios, sin especificar el tipo de los datos. En este ejemplo, por simplicidad, supondremos que todos los datos son enteros.

- En las líneas 9–10, se solicita al usuario que escriba por teclado el nombre del fichero que se desea leer, lo que no sería necesario si ya se conoce el nombre del fichero.

- En la línea 11, se abre el fichero cuyo nombre se indica como primer parámetro en el modo indicado como segundo parámetro[2]. El primer parámetro es una cadena de caracteres (`nombre`) que debe contener la ruta (relativa o absoluta) del fichero. El segundo parámetro es una cadena de caracteres donde el carácter `'r'` indica modo lectura (del inglés, *"read"*) y el carácter `'b'` indica formato binario (del inglés, *"binary"*.

- La línea 12 define el formato de un fichero. Aunque ahora no tengamos el tipo de dato fichero binario de tipo T, los ficheros generalmente tendrán una cierta estructura: ficheros de enteros, ficheros de reales, ficheros de estudiantes donde cada uno de ellas es una tripleta ⟨ cadena de caracteres, entero, real ⟩, etc. Por ejemplo (véase [9] para más opciones):

  - `'i'` indica entero (del inglés, *"integer"*),
  - `'f'`: indica real (del inglés, *"float"*),
  - `'Xs'`: cadena de caracteres de longitud X, por ejemplo "5s", que es importante de especificar en la práctica y
  - `'?'`: indica booleano.

  En realidad Python trata todos los ficheros binarios como una secuencia de bytes, pero dichos bytes podrán interpretarse de acuerdo con el formato especificado por el programador. Como veremos más adelante, el formato puede ser más complejo.

[2] Aunque el editor Pycharm avisa de que el tipo esperado `BufferedReader` no coincide con el tipo real `BinaryIO` (*"Expected type BufferedReader, got BinaryIO instead"*), el tipo real es verdaderamente `BinaryIO`.

- En la línea 13, se calcula el número de bytes que ocupa la estructura básica de cada fichero. En nuestro ejemplo, al haber usado un formato sencillo como ‘‘i’’ se correspondería con el tamaño en bytes de un entero (4).

- En las líneas 14–15 se define un bucle controlado por la variable booleana `eof`, inicialmente con valor `False` y que se repetirá mientras la variable siga teniendo valor `False`.

- En la línea 16, el procedimiento `read` lee un dato del fichero `f`, una lista de bytes del tamaño indicado en la variable `numBytes`, y lo almacena en la variable `misBytes`. `read` también avanza el puntero del fichero para que quede preparado para la lectura del siguiente dato. Si se ha llegado al final del fichero, `read` devuelve una cadena de caracteres vacía.

- La línea 17 comprueba si `misBytes` contiene una cadena vacía. En ese caso, la línea 18 asigna a la variable `eof` el valor `True` para poder salir del bucle while. En otro caso, las líneas 20–22 procesan el dato leído.

- En la línea 20 se decodifica la lista de bytes leídas del fichero, interpretándola según el formato indicado especificado por el programador. En nuestro caso, la lista de 4 bytes se interpreta como un número entero y en la línea 21 se asigna a la variable `dato`. La función `unpack_from` devuelve como resultado (`r`) una colección de valores, que podemos considerar un array[3], que en este caso es de tamaño uno. Más adelante veremos un ejemplo más complejo.

- La línea 22 escribe el dato por pantalla como ejemplo concreto de procesamiento del dato leído; en otros programas el procesamiento será diferente y, casi con total más seguridad, bastante más complejo.

- Tras haber salido del bucle de la línea 15 por haber llegado al final del fichero, la línea 23 cierra el fichero.

Para abrir un fichero de nombre conocido en tiempo de desarrollo del programa, podemos sustituir las líneas 9–10 del código anterior por una de las siguientes líneas, según prefiramos una ruta relativa o una ruta absoluta:

```
f = open('fichero.dat', 'rb')
f = open('C:\\Ejemplos␣Python\\fichero.dat', 'rb')
```

[3]En realidad, es de tipo `tuple`

Obsérvese que el carácter "\", que se usa cuando en las rutas aparece una carpeta, debe escribirse dos veces consecutivas y no solo una. Esto se conoce como "escapar" el carácter, e indica al compilar que no se va a usar ese símbolo para escribir un carácter especial (como el salto de línea "\n" o el tabulador "\t", sino para referirse realmente al propio carácter que representa la barra inclinada.

Después de este ejemplo relativamente sencillo, veamos cómo leer un fichero de estudiantes. Supondremos que para cada estudiante el fichero tiene 3 datos: una cadena de caracteres (de hasta 20, pero posiblemente menos) con el nombre, un identificador (ID) de tipo entero y una nota (número real, típicamente entre 0 y 10). El código sería:

```
1  import io
2  import struct
3  datoId: int
4  datoNombre: str
5  datoNota: float
6  eof: bool
7  f: io.BufferedReader
8  formato: struct
9  misBytes: bytes
10 f = open('estudiantes.dat', 'rb')
11 formato = struct.Struct('20s␣i␣f')
12 numBytes = struct.calcsize('20s␣i␣f')
13 eof = False
14 while not eof:
15     misBytes = f.read(numBytes)
16     if not misBytes:
17         eof = True
18     else:
19         r = formato.unpack_from(misBytes)
20         datoNombre = r[0].decode().rstrip('\x00')
21         datoId = r[1]
22         datoNota = r[2]
23         print(datoNombre, '␣', datoId, '␣', datoNota)
24 f.close()
```

Con respecto al código anterior en Pascal (donde leíamos un fichero de enteros), los cambios son los siguientes:

- Ahora necesitamos tres variables para almacenar los datos leídos: `datoId` (entero), `datoNombre` (cadena de caracteres), `datoNota` (real) (líneas 3–5).
- En lugar de solicitar al usuario el nombre del archivo, por simplicidad, hemos supuesto que es `'estudiantes.dat'` (línea 10).
- El formato es diferente: `'20s i f'` (líneas 11–12), es decir, una cadena de 20

caracteres (`'20s'`), un entero (`'i'`) y un real (`'f'`).

- La decodificación del dato leído es más compleja. Ahora, la estructura resultante de la llamada a `unpack_from` (`r`) es de tamaño 3. En la primera posición tenemos la cadena de caracteres. La función `decode` transforma de bytes a cadena de exactamente 20 caracteres, añadiendo a la cadena original tantos caracteres `'\x00'` como sean necesarios para que el nuevo tamaño de la cadena sea 20. Para solucionarlo, se invoca a la función `strip`, que elimina todas las ocurrencias del carácter `'\x00'` (línea 20). El procesamiento de los enteros, los reales y los booleanos es más sencillo: basta con acceder a las posiciones correspondientes del resultado devuelto por `unpack_from`.

- Evidentemente, a la hora de mostrar cada dato leído por pantalla, tenemos que escribir los tres datos de cada estudiante (línea 23).

10.4. Escritura de ficheros binarios

A continuación, veremos cómo escribir ficheros binarios. Entre otras cosas, nos servirá para poder probar los códigos desarrollados en el apartado anterior. Empezaremos por Pascal y, como primer ejemplo, veremos cómo crear un fichero de enteros a partir de los números que escriba el usuario por teclado:

```
1  PROGRAM escribirFichero;
2  TYPE
3     tipoDato = integer;
4     tipoFichero = FILE OF tipoDato;
5  VAR
6     dato : tipoDato;
7     f : tipoFichero;
8     nombre : STRING;
9  BEGIN
10    write('Escriba␣nombre␣del␣fichero:␣');
11    readln(nombre);
12    assign(f, nombre);
13    rewrite(f);
14    write('Escriba␣una␣lista␣de␣enteros␣(uno␣por␣linea):␣');
15    WHILE NOT eoln DO
16    BEGIN
17       readln(dato);  { Obtener un dato }
18       write(f, dato);
19    END;
20    close(f)
21 END.
```

Si comparamos este código con el de la Sección 10.3, podemos observar las siguientes diferencias:

- En la línea 13, el procedimiento `rewrite` indica que se va a trabajar en modo escritura con el fichero `f`. Si el fichero no existe, se crea. Si el fichero existe, se borra su contenido. En todo caso, el fichero queda preparado para la escritura y posicionado "al principio" del fichero[4].

- En la línea 14, se solicita al usuario que escriba una lista de enteros. El bucle de la línea 15 se repite mientras el usuario no introduzca una línea en blanco (sin ningún número). En cada iteración del bucle, se obtiene un dato. En este caso, simplemente se un entero de teclado (línea 17). Normalmente, el procesamiento necesario para obtener cada dato a escribir será diferente y significativamente más complejo.

- En la línea 18, el procedimiento write escribe en el fichero `f` el dato almacenado en la variable `dato`. `write`, que ya conocemos para escribir por pantalla el valor de una o varias variables, también permite escribir en un fichero si se le suministra como primer parámetro en la llamada al procedimiento una variable administrativa de fichero. Al igual que sucede con `read`, `write` permite escribir diferentes tipos de dato cuando trabaja con ficheros y cuando trabaja con la pantalla. Si tenemos un fichero binario de tipo T, `write` solamente puede escribir en el fichero datos de tipo T, sean o no tipos predefinidos.

  Una vez escrito un dato, `write` avanza automáticamente el puntero del fichero para que quede preparado para escribir los siguientes datos después de los datos actuales. Es decir, los datos estarán en el fichero en el mismo orden en el que los escribió el usuario durante la ejecución del programa.

¡Atención! 21. *Es muy importante no abrir en modo escritura un fichero cuyo contenido queremos conservar, porque se borraría su contenido.*

Supongamos ahora que queremos escribir un fichero de estudiantes, donde `tipoDato` es un alias de `tipoEstudiante`. A modo de ejemplo, vamos a escribir dos registros en el fichero, el primero correspondiente a Kelly Kapowski y el segundo a Zack Morris. Observemos que el orden en que asignamos los valores del nombre, la id y la nota no es relevante; lo único importante es que todos los valores del estudiante se definan antes de escribirlo a fichero:

[4]Lo entrecomillamos porque, en realidad, el fichero no tiene datos todavía: "programador, no hay fichero, se hace fichero al escribir".

```
dato.nombre := 'Kelly␣Kapowski';
dato.id := 123;
dato.nota := 9.5;
write(f, dato);
dato.id := 456;
dato.nota := 5.0;
dato.nombre := 'Zack␣Morris';
write(f, dato);
```

¡Atención! 22. *Son errores frecuentes:*

- *Asociar los ficheros pero no llamar a* `rewrite` *para iniciar la escritura*
- *Escribir sin abrir el fichero*
- *Escribir habiendo abierto el fichero en modo lectura*
- *Escribir un tipo de dato que no se corresponde con el tipo de dato del fichero*
- *Escribir en fichero un campo de un registro*

Ampliación 24. *En realidad, en Pascal también es posible trabajar con ficheros como si fueran secuencias de bytes. Veamos un ejemplo de código:*

```
1  PROGRAM leerBytes;
2  VAR
3     datos : ARRAY [1..4] OF byte;
4     f : FILE;
5     unEntero : integer;
6  BEGIN
7     assign(f, 'fichero.dat');
8     reset(f, 1);
9     WHILE NOT eof(f) DO
10    BEGIN
11       blockread(f, datos, sizeof(datos));
12       blockread(f, unEntero, sizeof(unEntero));
13       { Hacer algo con los datos }
14       { ... }
15    END;
16    close(f);
17 END.
```

Algunas consideraciones sobre este código:

- *Se puede utilizar simplemente el tipo* `FILE`*, sin especificar el tipo de los datos (que serán sencillamente bytes).*

- *En este caso, al iniciar la lectura con* **`reset`**, *es necesario añadir como segundo parámetro el tamaño por defecto de los bloques de bytes leídos; lo más habitual es usar* 1 *para leer byte a byte.*

- *En vez de usar* **`read`** *para leer datos, se usa* **`blockread`** *que, además de recibir una variable administrativa de fichero y la variable cuyo valor se quiere leer, recibe el número de bytes que se desean leer. Para obtenerlo, la función* **`sizeof`** *devuelve el tamaño en bytes de su argumento.*

- *Como vemos, se puede leer un array de bytes, en este caso de* 4 *bytes (línea* 11*), pero también es posible leer* 4 *bytes e interpretarlos como un número entero (línea* 12*).*

Veamos ahora los equivalentes en Python. Comencemos por el código para crear un fichero de enteros leídos por teclado:

```
1  import io
2  import struct
3  dato: int
4  f: io.BufferedWriter
5  formatoEntero: struct
6  leido: str
7  misBytes: bytes
8  nombre: str
9  seguir: bool
10 print("Escriba␣nombre␣del␣fichero:␣", end='')
11 nombre = input()
12 f = open(nombre, 'wb')
13 formatoEntero = struct.Struct('i')
14 print("Escriba␣una␣lista␣de␣enteros␣(uno␣por␣línea):␣")
15 seguir = True
16 while seguir:
17     leido = input()  # Obtener el dato
18     if leido == '':
19         seguir = False
20     else:
21         dato = int(leido)
22         misBytes = formatoEntero.pack(dato)
23         f.write(misBytes)
24 f.close()
```

Si comparamos este código con la lectura de un fichero de enteros, podemos observar las siguientes diferencias:

- Ahora se usa la el tipo de dato `BufferedWriter` (línea 4)[5], definido en el paquete `io`, para escribir ficheros binarios.
- La variable `leido` (línea 6) permite obtener números enteros (en este caso, a modo de ejemplo, de teclado) para escribirlos al fichero. Al leer de teclado (línea 17) su valor el de tipo cadena de caracteres, pero en la línea 21 se convierte a entero `dato`.
- Para cada número entero obtenido, se aplica la función `pack` al formato deseado (`formatoEntero`, de tipo `struct`) pasando como argumento el dato que se desea convertir a secuencia de bytes.
- Se invoca a la función `write` del fichero `f`, pasando como argumento la secuencia de bytes `misBytes`.
- El bucle se repite mientras queden números enteros por escribir, es decir, se repetirá tantas veces como números enteros escriba el usuario por teclado.

Para escribir un fichero de estudiantes, cada uno de los cuales contiene un nombre de hasta 20 carácteres, un identificador de tipo entero y una nota de tipo real, podemos usar el siguiente código:

```
1  import io
2  import struct
3  misBytes: bytes
4  f: io.BufferedWriter
5  formato: struct
6  f = open('estudiantes.dat', 'bw')
7  formato = struct.Struct('20s␣i␣f')
8  misBytes = formato.pack(
9      'Kelly␣Kapowski'.encode('utf-8'), 123, 9.5)
10 f.write(misBytes)
11 misBytes = formato.pack(
12     'Zack␣Morris'.encode('utf-8'), 456, 5.0)
13 f.write(misBytes)
14 f.close()
```

A modo de ejemplo, solo hemos añadido los datos de dos estudiantes, al igual que hicimos en Pascal. Con respecto al código anterior donde leíamos un fichero de enteros, el único cambio trascendente es la definición del formato '20s i f', tal y como se explicó cuando abordamos la lectura de un fichero de estudiantes.

[5]Aunque el editor Pycharm avisa de que el tipo esperado `BufferedWriter` no coincide con el tipo encontrado `BinaryIO`, (*Expected type BufferedWriter, got BinaryIO instead"*), el tipo usado es verdaderamente `BufferedWriter`.

Cabe subrayar que en Python, no es necesario elegir entre modo lectura (`'r'`) y escritura (`'w'`), sino que se puede usar un modo mixto (`'+'`). Además, `'a'` (del inglés *append*) permite anexar contenido al final del fichero.

10.5. Lectura de ficheros de texto

En esta sección estudiaremos cómo leer un fichero de texto carácter a carácter. Para trabajar con ficheros de texto en Pascal debemos tener cuenta algunas diferencias con respecto a los ficheros binarios:

- Para manejar ficheros de texto, las variables administrativas de fichero deben ser del tipo `TEXT`.
- Los ficheros de texto se estructuran en líneas. Para comprobar si se ha llegado al final de una línea, la función `eoln` devuelve `true` si el fichero que se pasa como único argumento ha llegado al final de la línea actual, o `false` en caso contrario.
- Cuando leemos de un fichero de texto, `read` funciona exactamente igual que al leer de teclado: se pueden leer caracteres, cadenas de caracteres, enteros o reales, pero no tipos de datos no predefinidos. Obsérvese que esto es diferente a lo que sucedía con los ficheros binarios, donde sí se podía leer un tipo de dato definido por el programador.
- Además de leer con `read`, podemos leer con `readln`, pasando como primer argumento una variable administrativa de fichero. `readln` funciona exactamente igual que al leer de teclado, leyendo el valor de las variables que se pasen como argumento y, además, leyendo hasta el siguiente salto de línea (incluido).

Ahora, para leer un fichero de texto, la idea es abrirlo en modo lectura, recorrer todas las líneas del fichero con un bucle y, para cada una de ellas, tener un segundo bucle anidado que lea todos los caracteres de la línea. Lo que distinguirá un problema de otro será el uso que se haga de esos datos.

¡Atención! 23. *Aunque los saltos de línea se codifican como caracteres, es preferible usar la función* `eoln` *para comprobar si se ha llegado al final de línea en lugar de comprobar si el último dato leído se corresponde con el carácter que codifica el fin de línea. El motivo es que el fin de línea puede codificarse de diferente manera dependiendo del sistema operativo, de hecho en Windows se codifica mediante dos caracteres (*"Carriage return"*, código ASCII* 13 *y* "Line feed"*, código ASCII* 10*), mientras que en macOs y Linux se usa un único carácter (*"Line feed"*).*

Veamos un ejemplo de código en Pascal. A modo de ejemplo, simplemente escribiremos cada dato leído por pantalla, de manera que se mostraría al usuario una copia exacta del contenido del fichero:

```
1  PROGRAM leerFicheroDeTexto;
2  VAR
3     dato : char;
4     f : text;
5     nombre : STRING;
6  BEGIN
7     write('Escriba␣nombre␣del␣fichero:␣');
8     readln(nombre);
9     assign(f, nombre);
10    reset(f);
11    WHILE NOT eof(f) DO
12    BEGIN
13       WHILE NOT eoln(f) DO
14       BEGIN
15          read(f, dato);
16          write(dato);  { Hacer algo con el dato }
17       END;
18       readln(f);
19       writeln
20    END;
21    close(f);
22 END.
```

Este código es bastante similar al que vimos en la Sección 10.3, por lo que los comentarios que hicimos sobre código son similares:

- El tipo de dato `text` usado en la línea 3 permite manejar ficheros de texto, así que la variable `f` permitirá manejar ficheros de texto.
- Las líneas 7–8 solicitan al usuario que escriba por teclado el nombre del fichero que se desea leer, lo que podría evitarse cuando se conozca el nombre del fichero.
- En la línea 9, `assign` vincula la variable administrativa de fichero `f` con el fichero cuyo nombre se indica mediante la cadena de caracteres `nombre`.
- En la línea 10, `reset` indica que se va a trabajar en modo lectura con `f` y se posiciona al principio del archivo.
- En la línea 11, `eof` devuelve un valor booleano que indica si se ha llegado al final del fichero o no.

- En la línea 13, la función `eoln` devuelve un valor booleano que indica si se ha llegado al final de la línea actual o no.
- En la línea 15, `read` lee un dato del fichero `f` y lo almacena en la variable `dato`.
- La línea 16 usa de alguna manera el dato leído (en este caso, simplemente lo escribe por pantalla).
- Una vez que se sale del bucle de la línea 13 porque se ha llegado al final de la línea, en la línea 18 se llama a `readln` para leer ese carácter de fin de línea y pasar a la siguiente.
- La línea 19 escribe un salto de línea por pantalla, para que el texto visualizado por pantalla sea igual que el contenido del fichero: como se acaba de leer un salto de línea de fichero, se escribe un salto de línea por pantalla.
- Una vez que se sale del bucle de la línea 11 porque se ha llegado al final del fichero, la línea 21 cierra el fichero.

Al igual que `read` y `readln` funcionan igual cuando se lee de teclado que cuando se lee de un fichero de texto; veremos más adelante que `write` y `writeln` también funcionan igual cuando se escribe por pantalla que cuando se escribe a un fichero de texto.

Ampliación 25. *De hecho, la entrada estándar* "input" *(el teclado) y la salida estándar* "output" *(la pantalla) pueden verse como dos ficheros especiales que siempre están abiertos en modo lectura y en modo escritura, respectivamente.*

Nótese que en Pascal, si se lee una variable de tipo carácter, `read` y `readln` leen un único carácter, pero si se lee una variable de tipo entero o real, se leen varios caracteres, puesto que se leen todos los dígitos consecutivos que se encuentren hasta llegar a un separador (espacio, tabulador o salto de línea).

Veamos ahora ejemplos más complicado, donde no se lea carácter a carácter. Supongamos que tenemos un fichero de texto donde en cada línea hay varios números enteros, separados por espacios, tabuladores o, en general, una secuencia de caracteres separadores que `read` sea capaz de procesar. El programa anterior seguiría siendo válido, con el único cambio de declarar `dato` como de tipo `integer`. Sin embargo, es importante mencionar que cuando se leen enteros o reales de un fichero de texto, si hay algún carácter separador tras el último dato, tendremos problemas: `eoln` no detectará el fin de línea (porque antes se encuentran esos caracteres separadores) y se intentará leer otro dato de la línea actual sin haber pasado a la siguiente.

Supongamos que tenemos un fichero de texto que codifica algunos datos de la tabla periódica de elementos químicos, con el siguiente formato: cada línea comienza con una letra mayúscula, que es la primera letra del símbolo del elemento. A continuación, opcionalmente, puede haber un espacio en blanco y una letra minúscula. El motivo es que el símbolo de algunos elementos químicos tiene solo una letra (por ejemplo, "H" para el hidrógeno) pero el de otros tiene dos letras (por ejemplo, "He" para el helio). En todo caso, después del último carácter del nombre del elemento, aparecerá un tabulador, después un entero representando el número atómico, otro tabulador, un número real representado la masa atómica del elemento y, por último, el salto de línea. Las dos primeras líneas del fichero tendrían el siguiente aspecto:

```
H    1    1.00794
H e  2    4.002602
```

Construyamos ahora un programa que lee una fichero con ese formato:

```
1  PROGRAM leerTexto;
2  VAR
3     c1, c2 : char;
4     f : text;
5     masaAtomica : real;
6     numAtomico : integer;
7  BEGIN
8     assign(f, 'periodica.txt');
9     reset(f);
10    WHILE NOT eof(f) DO
11    BEGIN
12       read(f, c1, c2);
13       IF c2 = '␣' THEN
14         read(f, c2);
15       readln(f, numAtomico, masaAtomica);
16       { Hacer algo con el dato ... }
17    END;
18    close(f);
19 END.
```

Para cada iteración del bucle (línea 10), que se repite hasta llegar al final del fichero, procedemos de la siguiente manera. En la línea 12, leemos un primer carácter `c1`, que se corresponderá seguro con la letra mayúscula del símbolo químico, y un segundo carácter `c2` que puede ser un tabulador o un espacio en blanco. Si es un espacio en blanco (línea 13), entonces se lee otro carácter `c2` (línea 14), que será la letra minúscula del símbolo químico y a continuación habrá un tabulador pendiente de haberse leído. En otro caso, ya se habrá leído el tabulador y se habrá almacenado en `c2`. Tanto en

un caso como en otro, al leer de fichero un entero `numAtomico` (línea 15), se leerá el número atómico, puesto que `read` sabe saltar los separadores previos al número entero. A continuación, la lectura de `masaAtomica` salta el tabulador previo, lee el número real de fichero y lo almacena en la variable. Por último, puesto que se ha usado `readln`, se lee el salto de línea, pasando a la siguiente línea si la hubiera.

¡Atención! 24. *Al usar* `eoln` *(igual que con* `eof`*,* `eoln` *y* `read`*) el funcionamiento es diferente si se pasa como parámetro un fichero o no. Si se nos olvida el parámetro, el compilador no avisa, porque es sintácticamente correcto no incluirlo.*

Veamos ahora cómo hacer códigos equivalentes en Python, comenzando por la lectura de un fichero de texto carácter a carácter:

```
1  import io
2  dato: str
3  eof: bool
4  eoln: bool
5  f: io.TextIOWrapper
6  nombre: str
7  print("Escriba␣nombre␣del␣fichero:␣", end='')
8  nombre = input()
9  f = open(nombre, 'rt')
10 eof = False
11 while not eof:
12     eoln = False
13     while not eoln:
14         dato = f.read(1)
15         if (dato == '\n') or (dato == ''):
16             eoln = True
17         else:
18             print(dato, end='')  # Hacer algo con el dato
19     if dato == '':
20         eof = True
21     else:
22         print()
23 f.close()
```

Expliquemos brevemente el código anterior:

- La línea 1 importa el paquete `io`, que será necesario para leer ficheros de texto
- Las líneas 2–6 declaran las variables necesarias. La línea 2 declara una variable para almacenar cada uno de los datos leídos, de tipo cadena de caracteres puesto que en Python no existe el tipo carácter. Las líneas 3–4 declaran variables booleanas para controlar los fines de línea y del fichero: `eof` indicará si se ha

llegado al final del fichero o no; `eoln` indicará si se ha llegado al final de línea o no. En la línea 5 se crea la variable administrativa de fichero de texto, de tipo `TextIOWrapper`[6], mientras que la línea 6 declara una variable de tipo cadena de caracteres para almacenar el nombre del fichero.

- Las líneas 7 y 8 piden al usuario que escriba por teclado el nombre del fichero a leer.

- La línea 9 abre el fichero con el nombre que se indica con el primer parámetro en el modo indicado por el segundo parámetro: `r` indica modo lectura (del inglés `read`) y `t` indica fichero de texto (del inglés `text`). En realidad, ambas opciones son los valores por defecto, por lo que no sería necesario indicarlas, pero al hacerlo el programa resulta mucho más legible.

- En la línea 10, se asigna valor inicial falso a `eof`. Si en algún momento se llega al final del fichero, su valor cambiará a verdadero. El bucle de la línea 11 se repite mientras el valor de `eof` sea falso.

- Para cada iteración del bucle, puesto que debe hacerse para cada nueva línea, la línea 12 asigna valor inicial falso a `eoln`. El bucle de la línea 13 se repite mientras el valor de `eoln` sea falso.

- En la línea 14 se lee un carácter del fichero `f` y se almacena en la variable `dato`, que en general será una cadena de caracteres de longitud uno, excepto si se llegó al fin del fichero, en cuyo caso será una cadena vacía.

- La línea 15 comprueba si el carácter leído en la línea anterior es un salto de línea o si se llegó al final del fichero sin que la última línea del fichero acabara en el carácter de fin de línea. En caso afirmativo, la línea 16 hace que la variable `eoln` cambie su valor a verdadero, de modo que la condición de parada del bucle de la línea 13 será falsa. En caso negativo, se procesará el dato. En nuestro ejemplo, el procesamiento del dato (en las líneas 17–18) únicamente consiste en escribir por pantalla el dato leído.

- Una vez que se ha acabado de procesar la línea del fichero, la línea 19 comprueba si se llegó al final del fichero y, en tal caso, la línea 20 hace que el valor de que la variable `eof` cambie a verdadero. En caso contrario, las líneas 21–22 escriben

[6] Aunque el editor Pycharm avisa de que el tipo esperado `TTextIOWrapper` no coincide con el tipo real `TextIO` (*"Expected type TextIOWrapper, got TextIO instead"*), el tipo real es verdaderamente `TextIOWrapper`.

un salto de línea para que la salida por pantalla se corresponda con el contenido del fichero.

- Finalmente, llegados al final del fichero, la línea 23 cierra el archivo.

Consideremos ahora el caso de la lectura de un fichero de texto con la tabla periódica. Para ello, podemos usar el siguiente código:

```
1  import io
2  c1: str
3  c2: str
4  eof: bool
5  f: io.TextIOWrapper
6  leido: list[str]
7  masaAtomica: float
8  numAtomico: int
9  f = open("periodica.txt", 'rt')
10 eof = False
11 while not eof:
12     c1 = f.read(1)
13     if c1 == '':
14         eof = True
15     else:
16         c2 = f.read(1)
17         if c2 == "␣":
18             c2 = f.read(1)
19             f.read(1)
20         leido = f.readline().split("\t")
21         numAtomico = int(leido[0])
22         masaAtomica = float(leido[1])
23         # Hacer algo con el dato
24         # ...
25 f.close()
```

La estructura del programa es similar al código anterior (lectura de un fichero de texto carácter a carácter): apertura del fichero, repetición de un bucle hasta el final del fichero y cierre del fichero. Lo que cambia radicalmente es el contenido del bucle:

- En la línea 12, `f.read(1)` lee un carácter de un fichero de texto (si fuera un fichero binario, leería un byte). Si el carácter leído es la cadena vacía, se ha llegado al final del fichero, así que la línea 14 ajusta la variable booleana para salir del bucle. En otro caso, será siempre la letra mayúscula del símbolo del elemento químico.

- En la línea 16, se lee otro carácter y se asigna a la variable `c2`, pudiendo ser un espacio en blanco o un tabulador. Si es un espacio en blanco, en la línea 18 se lee

otro carácter (que será la letra minúscula del símbolo) y en la línea 19 se lee otro carácter más (que será el tabulador), aunque en este último caso no se almacena en ninguna variable porque no lo necesitamos.

- Independientemente del número de letras en el símbolo del elemento químico, la línea 20 usa `readline` para leer el resto de la línea, que contendrá el número atómico, un tabulador y una masa atómica. A la cadena de caracteres devuelta por `readline` se le aplica la función `split`, que divide la cadena de caracteres de acuerdo con el carácter separador que se pase como parámetro (en este caso, el tabulador). Por lo tanto, devolverá una lista de 2 cadenas de caracteres: la primera de ellas incluirá el número atómico (por ejemplo, "1" ó "2") y la segunda la masa atómica (por ejemplo, "1.00794" ó "4.002602").

- Las dos cadenas de caracteres obtenidas al dividir al línea usando el tabulador se convierten a entero (línea 21) y a real (línea 22) para obtener el número atómico y la masa atómica, respectivamente.

Como diferencia importante con Pascal debemos destacar que `read` no es tan conveniente para leer un entero de fichero, ya que no sabemos a priori cuántos caracteres necesita; necesitaría leerse carácter a carácter y después reconstruir el valor numérico a partir de ellos o, como hemos hecho, leerse como cadena de caracteres y pasarse a número con `int` o `float`. Además, mientras que en Pascal `read` era capaz de saltar automáticamente los caracteres separadores cuando estábamos leyendo valores numéricos, en Python no es así, y hemos tenido que leer expresamente el tabulador que había tras la letra minúscula del elemento (línea 19).

10.6. Escritura de ficheros de texto

Una vez vistas las diferencias entre la lectura y la escritura de ficheros binarios, y entre la lectura de ficheros binarios y de texto, la escritura de ficheros de texto no tiene mucho misterio, pues combinaría las diferencias de ambos casos.

Probablemente el único aspecto que merece cierta atención es que, cuando escribimos en un fichero de texto, podemos usar tanto `write` como `writeln`, mientras que en ficheros binarios solo se puede usar `write`. Recordemos que, al igual que cuando escribimos por pantalla, la diferencia entre `write` y `writeln` es que este último, tras haber escrito en el fichero los datos pasados como parámetros, escribirá un salto de línea adicional. Escoger entre `write` y `writeln` depende del problema concreto (en particular, del formato de fichero deseado).

Veamos un ejemplo de escritura de fichero de texto en Pascal, a partir de los caracteres (posiblemente incluyendo saltos de línea) que el usuario escriba por teclado:

```
1  PROGRAM escribirFicheroTxt;
2  VAR
3     f : text;
4     nombre : STRING;
5     dato : STRING;
6  BEGIN
7     write('Escriba␣nombre␣del␣fichero:␣');
8     readln(nombre);
9     assign(f, nombre);
10    rewrite(f);
11    write('Escriba␣una␣lista␣de␣caracteres:␣');
12    WHILE NOT eoln DO
13    BEGIN
14       readln(dato); { Obtener datos }
15       writeln(f, dato);
16    END;
17    close(f)
18 END.
```

Para crear un fichero de texto con la tabla periódica, basta con aplicar los conocimientos sobre el uso de `write` a la hora de escribir caracteres, números y reales:

```
1  PROGRAM escribirTablaPeriodica;
2  VAR
3     f : text;
4  BEGIN
5     assign(f, 'periodica.txt');
6     rewrite(f);
7     writeln(f, 'H', chr(9), 1, char(9), 1.00794:1:5);
8     writeln(f, 'H␣e', chr(9), 2, char(9), 4.002602:1:6);
9     close(f)
10 END.
```

Recordemos que `chr(9)` devuelve el carácter asociado al entero 9 en la tabla ASCII, que es precisamente el tabulador. En la línea 7 se escribe el carácter "H", un tabulador, el entero 1, un tabulador y el real 1.00794, con 5 decimales. Por otro lado, en la línea 8 se escribe el carácter "H", un espacio en blanco, el carácter "e", un tabulador, el entero 2, un tabulador y el real 4.002602, con 6 decimales.

Por supuesto, el fichero se abre (línea 8), con las opciones `'wt'` para indicar modo escritura y formato fichero de texto. Para escribir en fichero, usamos `f.write` para escribir una cadena de caracteres en fichero. Concretamente, concatenamos la cadena de caracteres del usuario con un salto de línea, ya que en Python no existe una función

"`f.writeln`" que añadiera por sí misma el salto de línea final.

En Python, el equivalente al programa que escribe un fichero de texto a partir de la entrada del usuario sería:

```
from io import TextIOWrapper
f: TextIOWrapper
leido: str
nombre: str
seguir: bool
print("Escriba␣nombre␣del␣fichero:␣", end='')
nombre = input()
f = open(nombre, 'wt')
print("Escriba␣una␣lista␣de␣caracteres:␣")
seguir = True
while seguir:
    leido = input()  # Obtener datos
    if leido == '':
        seguir = False
    else:
        f.write(leido + '\n')
f.close()
```

Finalmente, el programa que escribe un fichero con la tabla periódica sería:

```
from io import TextIOWrapper
c1: str
c2: str
f: TextIOWrapper
numAtomico: int
masaAtomica: float
f = open("periodica.txt", 'wt')
c1 = 'H'
numAtomico = 1
masaAtomica = 1.00794
f.write(c1 + "\t" + str(numAtomico) + "\t" +
    str(masaAtomica) + "\n")
c1 = 'H'
c2 = 'e'
numAtomico = 2
masaAtomica = 4.002602
f.write(c1 + "␣" + c2 + "\t" + str(numAtomico) + "\t" +
    str(masaAtomica) + "\n")
f.close()
```

La escritura de las cadenas de caracteres, sean variables o constantes (`c1`, `c2`, el espacio en blanco y el tabulador) no precisa comentario alguno. Para escribir números (enteros o reales) o booleanos, la función `str` permite convertirlos a cadena de caracteres.

10.7. Ejercicios

Ejercicio 29. *Es frecuente que los programas informáticos utilicen ficheros de texto para codificar la geometría molecular de alguna sustancia química. Un posible formato se basa en indicar, para cada átomo de la molécula, sus coordenadas cartesianas en alguna unidad de medida (como el ångström). Por ejemplo, el siguiente fichero de texto* agua.txt *permite expresar la geometría del agua:*

```
O 0.000000 0.000000 0.117613
H 0.000000 0.757348 -0.470450
H 0.000000 -0.757348 -0.470450
```

Cada línea comienza con una letra mayúscula. Si el símbolo químico contiene dos letras, tras la letra mayúscula habrá otra minúscula (sin ningún espacio intermedio). Por simplicidad, descartaremos elementos químicos cuyo símbolo tenga más de dos letras. A continuación, seguirán un tabulador, un número real (coordenada X), un tabulador, un número real (coordenada Y), un tabulador, un número real (coordenada Z) y un salto de línea. Este formato está inspirado en uno de los formatos utilizados por el software de química computacional Gaussian09 (`http://www.gaussian.com`*), aunque requiere más parámetros.*

Escriba un programa que lea de teclado el nombre de un fichero de texto en el formato descrito y escriba por pantalla el número de átomos de cada elemento. Puede suponerse la constante `NUM_ELEMENTOS = 118`*. Por ejemplo, para el fichero* agua.txt*, la salida sería:* 1 atomo de O. 2 atomos de H.

Ejercicio 30. *Una tabla periódica de elementos químicos se puede representar como un valor del tipo de datos* `tablaPeriodica`*, consistente en un fichero secuencial de registros de tipo* `elemento`*. Un dato de tipo* `elemento` *almacena tres aspectos de un elemento químico como se declara a continuación en Pascal (o usando su equivalente en Python):*

```
CONST
    NUM_ELEMENTOS = 118;
TYPE
    elemento = RECORD
        nombre : STRING;
        simbolo : STRING[2];
        numeroAtomico : integer;
    END;
    tablaPeriodica = FILE OF elemento;
```

Construya un programa que escriba por pantalla si la tabla periódica suministrada en un fichero de tipo `tablaPeriodica` *y con nombre de fichero* tabla.dat*, es correcta o*

no. La tabla periódica se considerará que es correcta si y solo si:

- *contiene todos los elementos químicos cuyo número atómico sea inferior o igual a la constante* `NUM_ELEMENTOS`*;*
- *no incluye elementos repetidos, es decir, con el mismo número atómico; y*
- *no incluye elementos con número atómico superior a* `NUM_ELEMENTOS`*.*

Los registros almacenados en el fichero pueden aparecer desordenados y no seguir ningún criterio de orden de almacenamiento.

Ejercicio 31. *Considere el Ejercicio 30 y construya un programa en Pascal que use el fichero* tabla.dat *para calcular cual es la letra del alfabeto que aparece en los símbolos de más elementos químicos.*

Ejercicio 32. *Se desean comparar dos secuencias de material genético de virus SARS-CoV-2, obtenidas de dos pacientes diferentes, con el objetivo de analizar posibles mutaciones en el virus. Esto puede resultar útil para detectar diferentes cepas del virus (lo que puede resultar relevante para el diseño de vacunas) o para tratar de comprender mejor los mecanismos de expansión del virus (por ejemplo, identificar al paciente cero en una determinada región).*

El genoma del SARS-CoV-2 está compuesto del ácido nucleico ARN, que podemos ver por simplicidad como una secuencia de cuatro posibles bases nitrogenadas: adenina (A), la citosina (C), la guanina (G) y el uracilo (U). Como vimos en el Ejercicio 9, cada tres bases (cada codón) de una cadena corresponde a un aminoácido distinto. Por ejemplo, la asparagina puede codificarse mediante los codones 'AAC' y 'AAU'.

Se dispone de dos ficheros secuenciales (de tipo `fichero`*) y se desea comprobar la existencia de diferencias entre ellos. Se supone que ambos ficheros tienen exactamente el mismo número de datos, y que los únicos caracteres que aparecen son las iniciales de las cuatro bases nitrogenadas del ARN. Una diferencia entre los dos ficheros implica la existencia de una mutación. Además, esa mutación puede implicar un cambio de proteína o no. Por ejemplo, si en un fichero tenemos el codón 'AAC' y en otro el codón 'AAU', a pesar de haber una mutación en ambos casos se codifica la asparagina, pero si en el segundo fichero tuviéramos el codón 'AAA' la mutación implicaría un cambio de proteína, pues este último codón codifica la lisina.*

Construya una función que reciba los nombres de dos ficheros (de tipo `STRING`*) y que devuelva un valor que codifique los* 3 *posibles casos: si no ha habido mutaciones, si hay mutaciones pero no hay cambio de proteínas, o si hay mutaciones con cambio de proteínas.*

Se pueden suponer la existencia de una función `calculaAminoacido` *que dado un codón devuelve su aminoácido correspondiente y las siguientes definiciones de tipos (o su equivalente en Python):*

```
TYPE
    aminoacido = (AcidoAspartico, AcidoGlutamico, Alanina,
                  Arginina, Asparagina, Cisteina, Especial,
                  Fenilalanina, Glicina, Glutamina,
                  Histidina, Isoleucina, Leucina, Lisina,
                  Metionina, Prolina, Serina, Tirosina,
                  Treonina, Triptofano, Valina);
    codon = RECORD
        c1, c2, c3 : char;
    END;
    fichero = FILE OF codon;
```

Ejercicio 33. *Los parámetros de solubilidad de Hansen permiten predecir si una sustancia s_1 (soluto) se disolverá en otra sustancia s_2 (disolvente), basándose en la idea de que la disolución tendrá lugar entre moléculas similares. Concretamente, se trata de* 3 *parámetros: las energías debidas a las fuerzas de dispersión (δ_d), a las fuerzas polares (δ_p) y a los enlaces de hidrógeno δ_h entre moléculas. Estos parámetros pueden verse como las coordenadas de un punto en el espacio tridimensional, y cuanto más cerca estén dos moléculas entre sí, mayor es la probabilidad de que se disuelvan entre sí (de manera más precisa, ambas tendrían que estar dentro de la esfera definida por el radio de interacción del soluto R_1). En concreto, se trata de calcular la diferencia de energía relativa (RED)*

$$RED = \frac{\sqrt{4(\delta_{d2} - \delta_{d1})^2 + (\delta_{p2} - \delta_{p1})^2 + (\delta_{h2} - \delta_{h1})^2}}{R_1}$$

donde el subíndice $i \in \{1, 2\}$ indica que el parámetro corresponde a la sustancia s_i. Si la RED es menor que 1 *habrá disolución; si es igual a* 1 *habrá disolución parcial, y si es mayor que* 1 *no habrá disolución.*

Para caracterizar cada sustancia química, se usa un tipo de dato (`parametrosHansen`*) con los* 3 *parámetros de Hansen, el radio de interacción (en el caso de que la sustancia actúe como soluto) y un campo de texto con el nombre o la fórmula química. Se dispone de un fichero secuencial de registros que incluye los datos de varias sustancias.*

Implemente una función en Pascal que reciba una sustancia química (el soluto) y el nombre de otra (el disolvente) y devuelva la predicción acerca del tipo de disolución, consultando los datos del fichero hansen.dat.

Se suponen las siguientes definiciones de tipos (o su equivalente en Python):

```
TYPE
    disolucion = (disolucionTotal, disolucionParcial,
                  noDisolucion);
    parametrosHansen = RECORD
        fuerzasDispersion : real;
        fuerzasPolares : real;
        enlacesH : real;
        radioInteraccion : real;
        nombre : STRING;
    END;
    fichero = FILE OF parametrosHansen;
```

Bibliografía

[1] J. C. Barrera Juez and G. Romero de Tejada Muntaner. Motor de inteligencia artificial para un juego de mus. Treball de fi de carrera (undergraduate thesis project), Polytechnic University of Catalonia, Spain, 2008. http://hdl.handle.net/2099.1/5146.

[2] Free Pascal Online documentation. File handling functions. http://www.freepascal.org/docs-html/rtl/system/filefunctions.html, 2021.

[3] J. Gómez-Jurado. *Rey Blanco*. Ediciones B, 2021.

[4] Z. Kátai. *Algorythmics. Technologically and artistically enhanced computer science education*. Scientia Publishing House, 2021. http://real.mtak.hu/123062.

[5] R. McConnell, J. Haynes, and R. Warren. Understanding ASCII codes. http://www.nadcomm.com/ascii_code.htm, 1977.

[6] T. Oetiker, H. Partl, I. Hyna, and E. Schlegl. The not so short introduction to latex. http://tobi.oetiker.ch/lshort, 1994.

[7] Python Software Foundation. Built-in Types — Lists. http://docs.python.org/3/library/stdtypes.html#list, 2021.

[8] Python Software Foundation. Built-in Types — Text Sequence Type. http://docs.python.org/3/library/stdtypes.html#str, 2021.

[9] Python Software Foundation. struct — Interpret bytes as packed binary data. http://docs.python.org/3/library/struct.html, 2021.

[10] Python Software Foundation. math — Mathematical functions. http://docs.python.org/3/library/math.html, 2023.

[11] J. Smith. enum.Enum Functional API - regression in inspection and autocomplete. http://youtrack.jetbrains.com/issue/PY-54198/enum.Enum-Functional-API-regression-in-inspection-and-autocomplete, 2023.

[12] M. Van Canneyt. Reference guide for Free Pascal, version 3.2.2. http://www.freepascal.org/docs-html/ref/ref.html, 2021.

[13] G. Van Rossum and F. L. Drake. Python 3 Reference Manual, 2009.

[14] G. van Rossum, B. Warsaw, and N. Coghlan. PEP 8 – Style guide for Python code. http://www.python.org/dev/peps/pep-0008, 2001.