1.3. Programas y lenguajes de programación

Antes de detenernos a hablar de la codificación de la información estábamos comentando que la memoria es un gran almacén con cajones numerados, es decir, identificables con valores numéricos: sus respectivas direcciones. En cada cajón se almacena una secuencia de bits de tamaño fijo. La CPU, el «cerebro» del ordenador, es capaz de ejecutar acciones especificadas mediante secuencias de *instrucciones*. Una instrucción describe una acción muy simple, del estilo de «suma esto con aquello», «multiplica las cantidades que hay en tal y cual posición de memoria», «deja el resultado en tal dirección de memoria», «haz una copia del dato de esta dirección en esta otra dirección», «averigua si la cantidad almacenada en determinada dirección es negativa», etc. Las instrucciones se representan mediante combinaciones particulares de unos y ceros (valores binarios) y, por tanto, se pueden almacenar en la memoria.

Combinando inteligentemente las instrucciones en una secuencia podemos hacer que la CPU ejecute cálculos más complejos. Una secuencia de instrucciones es un *programa*. Si hay una instrucción para multiplicar pero ninguna para elevar un número al cubo, podemos construir un programa que efectúe este último cálculo a partir de las instrucciones disponibles. He aquí, grosso modo, una secuencia de instrucciones que calcula el cubo a partir de productos:

1. Toma el número y multiplícalo por sí mismo.
2. Multiplica el resultado de la última operación por el número original.

Las secuencias de instrucciones que el ordenador puede ejecutar reciben el nombre de *programas en código de máquina*, porque el *lenguaje de programación* en el que están expresadas recibe el nombre de *código de máquina*. Un lenguaje de programación es cualquier sistema de notación que permite expresar programas.

1.3.1. Código de máquina

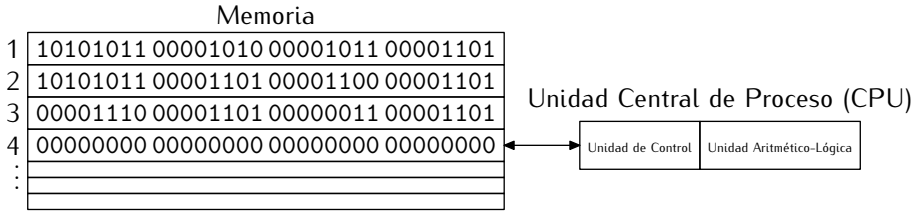
El código de máquina codifica las secuencias de instrucciones como sucesiones de unos y ceros que siguen ciertas reglas. Cada familia de ordenadores dispone de su propio repertorio de instrucciones, es decir, de su propio código de máquina.

Un programa que, por ejemplo, calcula la media de tres números almacenados en las posiciones de memoria 10, 11 y 12, respectivamente, y deja el resultado en la posición de memoria 13, podría tener el siguiente aspecto expresado de forma comprensible para nosotros:

Memoria

1	`Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13`
2	`Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13`
3	`Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13`
4	`Detener`

En realidad, el contenido de cada dirección estaría codificado como una serie de unos y ceros, así que el aspecto real de un programa como el descrito arriba podría ser éste:



La CPU es un ingenioso sistema de circuitos electrónicos capaz de interpretar el significado de cada una de esas secuencias de bits y llevar a cabo las acciones que codifican. Cuando la CPU ejecuta el programa empieza por la instrucción contenida en la primera de sus posiciones de memoria. Una vez ha ejecutado una instrucción, pasa a la siguiente, y sigue así hasta encontrar una instrucción que detenga la ejecución del programa.

Supongamos que en las direcciones de memoria 10, 11 y 12 se han almacenado los valores 5, 10 y 6, respectivamente. Representamos así la memoria:

Memoria

1	`Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13`
2	`Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13`
3	`Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13`
4	`Detener`
⋮	⋮
10	`5`
11	`10`
12	`6`
⋮	⋮

Naturalmente, los valores de las posiciones 10, 11 y 12 estarán codificados en binario, aunque hemos optado por representarlos en base 10 en aras de una mayor claridad.

La ejecución del programa procede del siguiente modo. En primer lugar, se ejecuta la instrucción de la dirección 1, que dice que tomemos el contenido de la dirección 10 (el valor 5), lo sumemos al de la dirección 11 (el valor 10) y dejemos el resultado (el valor 15) en la dirección de memoria 13. Tras ejecutar esta primera instrucción, la memoria queda así:

Memoria

1	`Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13`
2	`Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13`
3	`Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13`
4	`Detener`
⋮	⋮
10	`5`
11	`10`
12	`6`
13	`15`
⋮	⋮

A continuación, se ejecuta la instrucción de la dirección 2, que ordena que se tome el contenido de la dirección 13 (el valor 15), se sume al contenido de la dirección 12 (el valor 6) y se deposite el resultado (el valor 21) en la dirección 13. La memoria pasa a quedar en este estado.

Memoria

1	`Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13`
2	`Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13`
3	`Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13`
4	`Detener`
⋮	⋮
10	`5`
11	`10`
12	`6`
13	`21`
⋮	⋮

Ahora, la tercera instrucción dice que hemos de tomar el valor de la dirección 13 (el valor 21), dividirlo por 3 y depositar el resultado (el valor 7) en la dirección 13. Este es el estado en que queda la memoria tras ejecutar la tercera instrucción:

Memoria

1	`Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13`
2	`Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13`
3	`Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13`
4	`Detener`
⋮	⋮
10	`5`
11	`10`
12	`6`
13	`7`
⋮	⋮

Y finalmente, la CPU detiene la ejecución del programa, pues se encuentra con la instrucción **`Detener`** en la dirección 4.

EJERCICIOS

▶ 6 Ejecuta paso a paso el mismo programa con los valores 2, −2 y 0 en las posiciones de memoria 10, 11 y 12, respectivamente.

▶ 7 Diseña un programa que calcule la media de cinco números depositados en las posiciones de memoria que van de la 10 a la 14 y que deje el resultado en la dirección de memoria 15. Recuerda que la media $\bar{x}$ de cinco números x_1, x_2, x_3, x_4 y x_5 es

$$\bar{x} = \frac{\sum_{i=1}^{5} x_i}{5} = \frac{x_1 + x_2 + x_3 + x_4 + x_5}{5}.$$

▶ 8 Diseña un programa que calcule la varianza de cinco números depositados en las posiciones de memoria que van de la 10 a la 14 y que deje el resultado en la dirección de memoria 15. La varianza, que se denota con σ^2, es

$$\sigma^2 = \frac{\sum_{i=1}^{5} (x_i - \bar{x})^2}{5},$$

donde $\bar{x}$ es la media de los cinco valores. Supón que existe una instrucción «Multiplicar el contenido de dirección a por el contenido de dirección b y dejar el resultado en dirección c».

¿Qué instrucciones podemos usar para confeccionar programas? Ya hemos dicho que el ordenador sólo sabe ejecutar instrucciones muy sencillas. En nuestro ejemplo, sólo hemos utilizado tres instrucciones distintas:

- una instrucción de suma de la forma «**`Sumar contenido de direcciones` p `y` q `y dejar resultado en dirección` r**»;
- una instrucción de división de la forma «**`Dividir contenido de dirección` p `por` q `y dejar resultado en dirección` r**»;
- y una instrucción que indica que se ha llegado al final del programa: **`Detener`**.

¡Pocos programas interesantes podemos hacer con tan sólo estas tres instrucciones! Naturalmente, en un código de máquina hay instrucciones que permiten efectuar sumas, restas, divisiones y otras muchas operaciones. Y hay, además, instrucciones que permiten escoger qué instrucción se ejecutará a continuación, bien directamente, bien en función de si se cumple o no determinada condición (por ejemplo, «**`Si el último resultado es negativo, pasar a ejecutar la instrucción de la posición` p**»).

1.3.2. Lenguaje ensamblador

En los primeros tiempos de la informática los programas se introducían en el ordenador directamente en código de máquina, indicando uno por uno el valor de los bits de cada una de las posiciones de memoria. Para ello se insertaban manualmente cables en un panel de conectores: cada cable insertado en un conector representaba un uno y cada conector sin cable representaba un cero. Como puedes imaginar, programar así un computador resultaba una tarea ardua, extremadamente tediosa y propensa a la comisión de errores. El más mínimo fallo conducía a un programa incorrecto. Pronto se diseñaron notaciones que simplificaban la programación: cada instrucción de código de máquina se representaba mediante un *código mnemotécnico*, es decir, una abreviatura fácilmente identificable con el propósito de la instrucción.

Por ejemplo, el programa desarrollado antes se podría representar como el siguiente texto:

```
SUM #10, #11, #13
SUM #13, #12, #13
DIV #13, 3, #13
FIN
```

En este lenguaje la palabra **SUM** representa la instrucción de sumar, **DIV** la de dividir y **FIN** representa la instrucción que indica que debe finalizar la ejecución del programa. La almohadilla (**#**) delante de un número indica que deseamos acceder al contenido de la posición de memoria cuya dirección es dicho número. Los caracteres que representan el programa se introducen en la memoria del ordenador con la ayuda de un teclado y cada letra se almacena en una posición de memoria como una combinación particular de unos y ceros (su código ASCII, por ejemplo).

Pero, ¿cómo se puede ejecutar ese tipo de programa si la secuencia de unos y ceros que la describe como texto no constituye un programa válido en código de máquina? Con la ayuda de otro programa: el *ensamblador*. El ensamblador es un programa traductor que lee el contenido de las direcciones de memoria en las que hemos almacenado códigos mnemotécnicos y escribe en otras posiciones de memoria sus instrucciones asociadas en código de máquina.

El repertorio de códigos mnemotécnicos traducible a código de máquina y las reglas que permiten combinarlos, expresar direcciones, codificar valores numéricos, etc., recibe el nombre de *lenguaje ensamblador*, y es otro lenguaje de programación.

1.3.3. ¿Un programa diferente para cada ordenador?

Cada CPU tiene su propio juego de instrucciones y, en consecuencia, un código de maquina y uno o más lenguajes ensambladores propios. Un programa escrito para una CPU de la marca Intel no funcionará en una CPU diseñada por otro fabricante, como Motorola[3]. ¡Incluso diferentes versiones de una misma CPU tienen juegos de instrucciones que no son totalmente compatibles entre sí!: los modelos más evolucionados de una familia de CPU pueden incorporar instrucciones que no se encuentran en los más antiguos[4].

Si queremos que un programa se ejecute en más de un tipo de ordenador, ¿habrá que escribirlo de nuevo para cada CPU particular? Durante mucho tiempo se intentó definir algún tipo de «lenguaje ensamblador universal», es decir, un lenguaje cuyos códigos

[3] A menos que la CPU se haya diseñado expresamente para reproducir el funcionamiento de la primera, como ocurre con los procesadores de AMD, diseñados con el objetivo de ejecutar el código de máquina de los procesadores de Intel.

[4] Por ejemplo, añadiendo instrucciones que faciliten la programación de aplicaciones multimedia (como ocurre con los Intel Pentium MMX y modelos posteriores) impensables cuando se diseñó la primera CPU de la familia (el Intel 8086).

¡Hola, mundo!

Nos gustaría mostrarte el aspecto de los programas escritos en lenguajes ensambladores reales con un par de ejemplos. Es una tradición ilustrar los diferentes lenguajes de programación con un programa sencillo que se limita a mostrar por pantalla el mensaje «Hello, World!» («¡Hola, mundo!»), así que la seguiremos. He aquí ese programa escrito en los lenguajes ensambladores de dos CPU distintas: a mano izquierda, el de los procesadores 80x86 de Intel (cuyo último representante por el momento es el Pentium 4) y a mano derecha, el de los procesadores de la familia Motorola 68000 (que es el procesador de los primeros ordenadores Apple Macintosh).

```
        .data
    msg:
        .string "Hello, World!\n"
    len:
        .long . - msg
    .text
    .globl _start
    _start:
        push $len
        push $msg
        push $1
        movl $0x4, %eax
        call _syscall
        addl $12, %esp
        push $0
        movl $0x1, %eax
        call _syscall
    _syscall:
        int  $0x80
        ret
```

```
start:
  move.l #msg,-(a7)
  move.w #9,-(a7)
  trap #1
  addq.l #6,a7
  move.w #1,-(a7)
  trap #1
  addq.l  #2,a7
  clr -(a7)
  trap #1
  msg: dc.b "Hello, World!",10,13,0
```

Como puedes ver, ambos programas presentan un aspecto muy diferente. Por otra parte, los dos son bastante largos (entre 10 y 20 líneas) y de difícil comprensión.

mnemotécnicos, sin corresponderse con los del código de máquina de ningún ordenador concreto, fuesen fácilmente traducibles al código de máquina de cualquier ordenador. Disponer de dicho lenguaje permitiría escribir los programas una sola vez y ejecutarlos en diferentes ordenadores tras efectuar las correspondientes traducciones a cada código de máquina con diferentes programas ensambladores.

Si bien la idea es en principio interesante, presenta serios inconvenientes:

- Un lenguaje ensamblador universal no puede tener en cuenta cómo se diseñarán ordenadores en un futuro y qué tipo de instrucciones soportarán, así que posiblemente quede obsoleto en poco tiempo.
- Programar en lenguaje ensamblador (incluso en ese supuesto lenguaje ensamblador universal) es complicadísimo por los numerosos detalles que deben tenerse en cuenta.

Además, puestos a diseñar un lenguaje de programación general, ¿por qué no utilizar un lenguaje natural, es decir un lenguaje como el castellano o el inglés? Programar un computador consistiría, simplemente, en escribir (¡o pronunciar frente a un micrófono!) un texto en el que indicásemos qué deseamos que haga el ordenador usando el mismo lenguaje con que nos comunicamos con otras personas. Un programa informático podría encargarse de traducir nuestras frases al código de máquina, del mismo modo que un programa ensamblador traduce lenguaje ensamblador a código de máquina. Es una idea atractiva, pero que queda lejos de lo que sabemos hacer por varias razones:

- La complejidad intrínseca de las construcciones de los lenguajes naturales dificulta enormemente el *análisis sintáctico* de las frases, es decir, comprender su estructura y cómo se relacionan entre sí los diferentes elementos que las constituyen.
- El *análisis semántico*, es decir, la comprensión del significado de las frases, es aún más complicado. Las ambigüedades e imprecisiones del lenguaje natural hacen que sus frases presenten, fácilmente, diversos significados, aun cuando las podamos analizar sintácticamente. (¿Cuántos significados tiene la frase «Trabaja en un banco.»?) Sin una buena comprensión del significado no es posible efectuar una traducción aceptable.

1.3.4. Lenguajes de programación de alto nivel

Hay una solución intermedia: podemos diseñar lenguajes de programación que, sin ser tan potentes y expresivos como los lenguajes naturales, eliminen buena parte de la complejidad propia de los lenguajes ensambladores y estén bien adaptados al tipo de problemas que podemos resolver con los computadores: los denominados *lenguajes de programación de alto nivel*. El calificativo «de alto nivel» señala su independencia de un ordenador concreto. Por contraposición, los códigos de máquina y los lenguajes ensambladores se denominan *lenguajes de programación de bajo nivel*.

He aquí el programa que calcula la media de tres números en un lenguaje de alto nivel típico (Python):

```
a = 5
b = 10
c = 6
media = (a + b + c) / 3
```

Las tres primeras líneas definen los tres valores y la cuarta calcula la media. Como puedes ver, resulta mucho más legible que un programa en código de máquina o en un lenguaje ensamblador.

Para cada lenguaje de alto nivel y para cada CPU se puede escribir un programa que se encargue de traducir las instrucciones del lenguaje de alto nivel a instrucciones de código de máquina, con lo que se consigue la deseada independencia de los programas con respecto a los diferentes sistemas computadores. Sólo habrá que escribir una versión del programa en un lenguaje de programación de alto nivel y la traducción de ese programa al código de máquina de cada CPU se realizará automáticamente.

1.3.5. Compiladores e intérpretes

Hemos dicho que los lenguajes de alto nivel se traducen automáticamente a código de máquina, sí, pero has de saber que hay dos tipos diferentes de traductores dependiendo de su modo de funcionamiento: *compiladores* e *intérpretes*.

Un *compilador* lee completamente un programa en un lenguaje de alto nivel y lo traduce en su integridad a un programa de código de máquina equivalente. El programa de código de máquina resultante se puede ejecutar cuantas veces se desee, sin necesidad de volver a traducir el programa original.

Un *intérprete* actúa de un modo distinto: lee un programa escrito en un lenguaje de alto nivel instrucción a instrucción y, para cada una de ellas, efectúa una traducción a las instrucciones de código de máquina equivalentes y las ejecuta inmediatamente. No hay un proceso de traducción separado por completo del de ejecución. Cada vez que ejecutamos el programa con un intérprete, se repite el proceso de traducción y ejecución, ya que ambos son simultáneos.

Compiladores e intérpretes... de idiomas

Puede resultarte de ayuda establecer una analogía entre compiladores e intérpretes de lenguajes de programación y traductores e intérpretes de idiomas.

Un compilador actúa como un traductor que recibe un libro escrito en un idioma determinado (lenguaje de alto nivel) y escribe un nuevo libro que, con la mayor fidelidad posible, contiene una traducción del texto original a otro idioma (código de máquina). El proceso de traducción (compilación) tiene lugar una sola vez y podemos leer el libro (ejecutar el programa) en el idioma destino (código de máquina) cuantas veces queramos.

Un intérprete de programas actúa como su homónimo en el caso de los idiomas. Supón que se imparte una conferencia en inglés en diferentes ciudades y un intérprete ofrece su traducción simultánea al castellano. Cada vez que la conferencia es pronunciada, el intérprete debe realizar nuevamente la traducción. Es más, la traducción se produce sobre la marcha, frase a frase, y no de un tirón al final de la conferencia. Del mismo modo actúa un intérprete de un lenguaje de programación: traduce cada vez que ejecutamos el programa y además lo hace instrucción a instrucción.

Por regla general, los intérpretes ejecutarán los programas más lentamente, pues al tiempo de ejecución del código de máquina se suma el que consume la traducción simultánea. Además, un compilador puede examinar el programa de alto nivel abarcando más de una instrucción cada vez, por lo que es capaz de producir mejores traducciones. Un programa interpretado suele ser mucho más lento que otro equivalente que haya sido compilado (¡típicamente entre 2 y 100 veces más lento!).

Si tan lento resulta interpretar un programa, ¿por qué no se usan únicamente compiladores? Es pronto para que entiendas las razones, pero, por regla general, los intérpretes permiten una mayor flexibilidad que los compiladores y ciertos lenguajes de programación de alto nivel han sido diseñados para explotar esa mayor flexibilidad. Otros lenguajes de programación, por contra, sacrifican la flexibilidad en aras de una mayor velocidad de ejecución. Aunque nada impide que compilemos o interpretemos cualquier lenguaje de programación, ciertos lenguajes se consideran apropiados para que la traducción se lleve a cabo con un compilador y otros no. Es más apropiado hablar, pues, de lenguajes de programación *típicamente* interpretados y lenguajes de programación *típicamente* compilados. Entre los primeros podemos citar Python, BASIC, Perl, Tcl, Ruby, Bash, Java o Lisp. Entre los segundos, C, C#, Pascal, C++ o Fortran.

En este curso aprenderemos a programar usando dos lenguajes de programación distintos: uno interpretado, Python, y otro compilado, C. Este volumen se dedica al lenguaje de programación con Python. Otro volumen de la misma colección se dedica al estudio de C, pero partiendo de la base de que ya se sabe programar con Python.

1.3.6. Python

Existen muchos otros lenguajes de programación, ¿por qué aprender Python? Python presenta una serie de ventajas que lo hacen muy atractivo, tanto para su uso profesional como para el aprendizaje de la programación. Entre las más interesantes desde el punto de vista didáctico tenemos:

- Python es un lenguaje muy *expresivo*, es decir, los programas Python son muy compactos: un programa Python suele ser bastante más corto que su equivalente en lenguajes como C. (Python llega a ser considerado por muchos un *lenguaje de programación de muy alto nivel*.)
- Python es muy *legible*. La sintaxis de Python es muy elegante y permite la escritura de programas cuya lectura resulta más fácil que si utilizáramos otros lenguajes de programación.

- Python ofrece un *entorno interactivo* que facilita la realización de pruebas y ayuda a despejar dudas acerca de ciertas características del lenguaje.
- El *entorno de ejecución* de Python *detecta muchos de los errores* de programación que escapan al control de los compiladores y proporciona información muy rica para detectarlos y corregirlos.
- Python puede usarse como lenguaje *imperativo procedimental* o como lenguaje *orientado a objetos*.
- Posee un *rico juego de estructuras de datos* que se pueden manipular de modo sencillo.

Estas características hacen que sea relativamente fácil traducir métodos de cálculo a programas Python.

Python ha sido diseñado por Guido van Rossum y está en un proceso de continuo desarrollo por una gran comunidad de desarrolladores. Aproximadamente cada seis meses se hace pública una nueva versión de Python. ¡Tranquilo! No es que cada medio año se cambie radicalmente el lenguaje de programación, sino que éste se enriquece manteniendo en lo posible la compatibilidad con los programas escritos para versiones anteriores. Nosotros utilizaremos características de la versión 2.3 de Python, por lo que deberás utilizar esa versión o una superior.

Una ventaja fundamental de Python es la gratuidad de su intérprete. Puedes descargar el intérprete de la página web http://www.python.org. El intérprete de Python tiene versiones para prácticamente cualquier plataforma en uso: sistemas PC bajo Linux, sistemas PC bajo Microsoft Windows, sistemas Macintosh de Apple, etc.

Para que te vayas haciendo a la idea de qué aspecto presenta un programa completo en Python, te presentamos uno que calcula la media de tres números que introduce por teclado el usuario y muestra el resultado por pantalla:

```
a = float(raw_input('Dame un número:'))
b = float(raw_input('Dame otro número:'))
c = float(raw_input('Y ahora, uno más:'))
media = (a + b + c) / 3
print 'La media es', media
```

En los últimos años Python ha experimentado un importantísimo aumento del número de programadores y empresas que lo utilizan. Aquí tienes unas citas que han encabezado durante algún tiempo la web oficial de Python (http://www.python.org):

> *Python ha sido parte importante de Google desde el principio, y lo sigue siendo a medida que el sistema crece y evoluciona. Hoy día, docenas de ingenieros de Google usan Python y seguimos buscando gente diestra en este lenguaje.*
>
> PETER NORVIG, *director de calidad de búsquedas de Google Inc.*

> *Python juega un papel clave en nuestra cadena de producción. Sin él, un proyecto de la envergadura de «Star Wars: Episodio II» hubiera sido muy difícil de sacar adelante. Visualización de multitudes, proceso de lotes, composición de escenas... Python es lo que lo une todo.*
>
> TOMMY BRUNETTE, *director técnico senior de Industrial Light & Magic .*

Python está en todas partes de Industrial Light & Magic. Se usa para extender la capacidad de nuestras aplicaciones y para proporcionar la cola que las une. Cada imagen generada por computador que creamos incluye a Python en algún punto del proceso.

PHILIP PETERSON, *ingeniero principal de I+D de Industrial Light & Magic.*

1.3.7. C

El lenguaje de programación C es uno de los más utilizados en el mundo profesional. La mayoría de las aplicaciones comerciales y libres se han desarrollado con el lenguaje de programación C. El sistema operativo Linux, por ejemplo, se ha desarrollado en C en su práctica totalidad.

¿Por qué es tan utilizado el lenguaje C? C es un lenguaje de propósito general que permite controlar con gran precisión los factores que influyen en la eficiencia de los programas. Pero esta capacidad de control «fino» que ofrece C tiene un precio: la escritura de programas puede ser mucho más costosa, pues hemos de estar pendientes de numerosos detalles. Tan es así que muchos programadores afirman que C no es un lenguaje de alto nivel, sino de *nivel intermedio*.

¡Hola de nuevo, mundo!

Te presentamos los programas «¡Hola, mundo!» en Python (izquierda) y C (derecha).

```
print 'Hello, world!'
```

```
#include <stdio.h>

int main(void) {
   printf("Hello, world!\n");
   return 0;
}
```

Como puedes comprobar, Python parece ir directamente al problema: una sola línea. Empezaremos aprendiendo Python.

Aquí tienes una version en C del cálculo de la media de tres números leídos por teclado:

```
#include <stdio.h>

int main(void)
{
  double a, b, c, media;

  printf("Dame un número: ");
  scanf("%lf", &a);
  printf("Dame otro número: ");
  scanf("%lf", &b);
  printf("Y ahora, uno más: ");
  scanf("%lf", &c);
  media = (a + b + c) / 3;
  printf("La media es %f\n", media);
  return 0;
}
```

C ha sufrido una evolución desde su diseño en los años 70. El C, tal cual fue concebido por sus autores, Brian Kernighan y Dennis Ritchie, de la compañía norteamericana de

telecomunicaciones AT&T, se conoce popularmente por K&R C y está prácticamente en desuso. En los años 80, C fue modificado y estandarizado por el American National Standards Institute (ANSI), que dio lugar al denominado ANSI C y que ahora se conoce como C89 por el año en que se publicó. El estándar se revisó en los años 90 y se incorporaron nuevas características que mejoran sensiblemente el lenguaje. El resultado es la segunda edición del ANSI C, más conocida como C99. Esta es la versión que estudiaremos en este curso.

En la asignatura utilizaremos un compilador de C gratuito: el `gcc` en su versión 3.2 o superior. Inicialmente se denominó a `gcc` así tomando las siglas de GNU C Compiler. GNU es el nombre de un proyecto que tiene por objeto ofrecer un sistema operativo «libre» y todas las herramientas que es corriente encontrar en una plataforma Unix. Hoy día se ha popularizado enormemente gracias a la plataforma GNU/Linux, que se compone de un núcleo de sistema operativo de la familia del Unix (Linux) y numerosas herramientas desarrolladas como parte del proyecto GNU, entre ellas `gcc`. La versión de `gcc` que usaremos no soporta aún todas las características de C99, pero sí las que aprenderemos en el curso.

Cualquier distribución reciente de Linux[5] incorpora la versión de `gcc` que utilizaremos o una superior. Puedes descargar una versión de `gcc` y las utilidades asociadas para Microsoft Windows en `http://www.delorie.com/djgpp`. En la página `http://www.bloodshed.net/devcpp.html` encontrarás un entorno integrado (editor de texto, depurador de código, compilador, etc.) que también usa el compilador `gcc`.

¡Ojo!, no todos los compiladores soportan algunas características de la última versión de C, así que es posible que experimentes algún problema de compatibilidad si utilizas un compilador diferente del que te recomendamos.

1.4. Más allá de los programas: algoritmos

Dos programas que resuelven el mismo problema expresados en el mismo o en diferentes lenguajes de programación pero que siguen, en lo fundamental, el mismo procedimiento, son dos *implementaciones* del mismo *algoritmo*. Un algoritmo es, sencillamente, una secuencia de pasos orientada a la consecución de un objetivo.

Cuando diseñamos un algoritmo podemos expresarlo en uno cualquiera de los numerosos lenguajes de programación de propósito general existentes. Sin embargo, ello resulta poco adecuado:

- no todos los programadores conocen todos los lenguajes y no hay consenso acerca de cuál es el más adecuado para expresar las soluciones a los diferentes problemas,
- cualquiera de los lenguajes de programación presenta particularidades que pueden interferir en una expresión clara y concisa de la solución a un problema.

Podemos expresar los algoritmos en lenguaje natural, pues el objetivo es comunicar un procedimiento resolutivo a otras personas y, eventualmente, traducirlos a algún lenguaje de programación. Si, por ejemplo, deseamos calcular la media de tres números leídos de teclado podemos seguir este algoritmo:

1. solicitar el valor del primer número,
2. solicitar el valor del segundo número,
3. solicitar el valor del tercer número,

[5] En el momento en que se redactó este texto, las distribuciones más populares y recientes de Linux eran SuSE 8.2, RedHat 9, Mandrake 9.1 y Debian Woody.