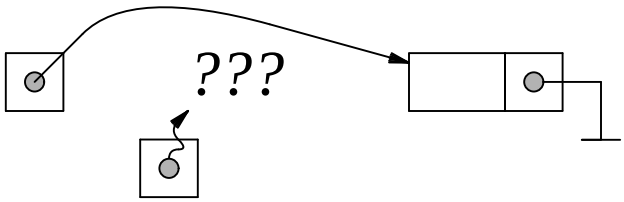
# *Curso Práctico de Programación*




# *Usando Picky como Primer Lenguaje*

*Francisco J Ballesteros*

*Enrique Soriano*

*Gorka Guardiola*

# 1 — Introducción a la programación

## 1.1. ¿Qué es programar?

Al contrario que otras máquinas, el ordenador puede hacer cualquier cosa que queramos si le damos las instrucciones oportunas. A darle estas instrucciones se le denomina **programar** el ordenador. Un ordenador es una máquina que tiene básicamente la estructura que puede verse en la figura 1.1. Posee algo de memoria para almacenar las instrucciones que le damos y tiene una unidad central para procesarlas (llamada CPU). En realidad, las instrucciones sirven para manipular información que también se guarda en la memoria del ordenador. A esta información la denominamos **datos** y a las instrucciones **programa**. Como la memoria no suele bastar, y además el contenido de la memoria se pierde cuando apagamos el ordenador, se usa un disco en el que se puede almacenar tanta información (programas incluidos) como se desee de forma permanente.

Utilizando una analogía, podemos decir que un programa no es otra cosa que una receta de cocina, los datos no son otra cosa que los ingredientes para seguir una receta y la CPU es un pinche de cocina extremadamente tonto, pero extremadamente obediente. Nosotros somos el Chef. Dependiendo de las órdenes que le demos al pinche podemos obtener o un exquisito postre o un armagedón culinario.

Es importante remarcar que la memoria del ordenador es volátil, esto es, se borra cada vez que apagamos el ordenador. La unidad central de proceso o **CPU** del ordenador es capaz de leer un programa ya almacenado en la memoria y de efectuar las acciones que dicho programa indica. Este programa puede manipular sólo datos almacenados también en la memoria. Esto no es un problema, puesto que existen programas en el ordenador que saben cómo escribir en la memoria los programas que queremos ejecutar y los datos que necesitan. A este último conjunto de programas se le suele llamar **sistema operativo**, puesto que es el sistema que nos deja operar con el ordenador.

El disco es otro tipo de almacenamiento para información similar a la memoria, aunque al contrario que ésta su contenido persiste mientras el ordenador está apagado. Su tamaño suele ser mucho mayor que el de la memoria del ordenador, por lo que se utiliza para mantener aquellos datos y programas que no se están utilizando en un momento determinado. Los elementos que guardan la información en el disco se denominan **ficheros** y no son otra cosa que nombres asociado a una serie de datos que guardamos en el disco (de ahí que se llamen ficheros, por analogía a los ficheros o archivos que empleamos en una oficina).

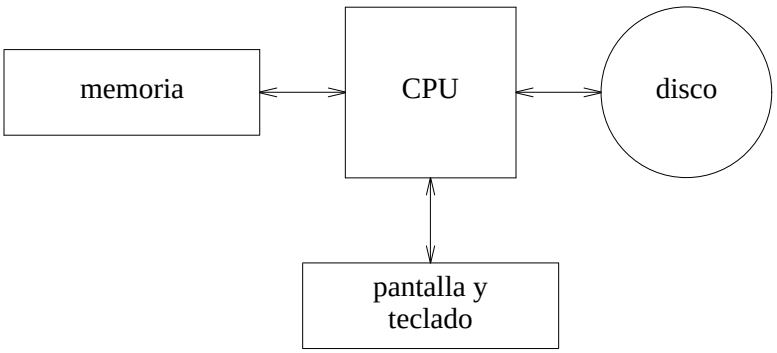



**Figura 1.1:** *Esquema de un ordenador.*

Tanto la memoria como el disco son capaces sólo de guardar información escrita como una serie de números. Estos números se escriben en base 2 o **binario**. Esto es, sólo podemos guardar unos y ceros en la memoria. Poder guardar sólo unos y ceros no es un problema. Con ellos podemos

representar cualquier cosa. Imagina un objeto que puede ser persona, animal, planta o cosa. Si me dejas hacerte suficientes preguntas que se respondan con ‘‘sí’’ o ‘‘no’’, es cuestión de hacer las preguntas adecuadas para saber en qué estas pensando. Si decimos que un 1 es un ‘‘sí’’ y un 0 es un ‘‘no’’ podríamos escribir tantos unos y ceros como respuestas afirmativas y negativas conduzcan a identificar el objeto en que se pensaba. Pues bien, toda esa secuencia de unos y ceros podrían ser la forma de representar este objeto en la memoria del ordenador.

Naturalmente la información no se guarda así. En lugar de eso se han buscado formas mejores de representarla. Por ejemplo, a cada una de las 256 operaciones básicas que podría saber hacer un ordenador le podemos asignar un número del 0 a 255 (ten en cuenta que si empezamos a asignar números desde el 0, al último elemento de los 256 elementos le corresponderá el número 255). Si le damos uno de estos números a la CPU, ella se ocupará de hacer la operación correspondiente. A lo mejor 15 significa sumar y 18 restar. Igualmente podemos hacer para almacenar texto escrito en el ordenador. Podemos utilizar un número distinto para cada letra y guardarlos todos seguidos en la memoria según las letras que queramos escribir. Todo son números. Lo que signifique cada número depende en realidad del programa que lo utiliza (y de la persona que escribe el programa).

Al conjunto de programas que hay en un ordenador se le denomina **software** (literalmente *parte blanda*, que se refiere a los componentes no tangibles). Al conjunto físico de cables, plástico, metal y circuitos se le denomina **hardware** (literalmente *parte dura* o tangible). Programar el ordenador consiste en introducirle un programa que efectúe las acciones que deseamos para resolver un problema. Dicho de otra forma, programar es construir software.

**Ejecutar** un programa es pedirle al ordenador que siga los pasos que indica dicho programa. En realidad esto se lo tenemos que pedir al sistema operativo, que se ocupará de escribir (cargar) en la memoria el programa que queremos ejecutar, tras lo cual la CPU hará el resto.

Antes de ver cómo programar, necesitamos conocer los nombres que se utilizan para medir cantidades de memoria. A una respuesta de tipo sí/no, esto es, a un dígito binario, se le denomina **bit** (por *binary digit*). Un número de 8 bits se denomina **byte**. Este tamaño tiene un nombre especial por su alta popularidad (por ejemplo, podemos utilizar un byte distinto para representar cada letra de un texto). A 1024 bytes se les suele llamar **Kilobyte** o **Kb**. Se supone que en realidad deberíamos decir *Kibibyte* y escribirlo *KiB*, pero en la práctica se sigue usando el término Kilobyte, aunque cada vez se usa más el símbolo *KiB*. 1024 Kilobytes hacen un **Megabyte** o **MB**. Igual que antes, debería decirse *Mebibyte*, cuyo símbolo es *MiB*, pero en la práctica se sigue usando el término Megabyte (aunque no es raro ver el símbolo *MiB*). 1024 Mbytes constituyen un **Gigabyte** o **GB** (el término preciso es *Gibibyte* y su símbolo es *GiB*). Y así sucesivamente tenemos **Terabyte**, **Petabyte** y **Exabyte**.

La memoria de un ordenador suele ser de unos pocos Gigabytes. Los discos actuales suelen rondar el Terabyte. Todo esto nos importa para poder expresar cuánta memoria consume un programa o qué tamaño tiene el programa en si mismo, o cuánto almacenamiento necesitamos para los datos.

## 1.2. Programa para programar

En realidad programar es describir un plan con sumo detalle y empleando un lenguaje que luego el ordenador pueda entender. En este epígrafe vamos a ver cómo podemos proceder para construir programas. Lo podríamos denominar un plan para programar. Lamentablemente, el hardware es tan tonto que no sabría qué hacer con un plan expresado en lenguaje natural (español). Pero a nosotros este plan nos puede servir perfectamente. Los pasos necesarios para realizar un programa son los siguientes:

1 **Definir el problema**. En primer lugar hay que tener muy claro qué es lo que se pretende resolver. Si no sabemos qué queremos hacer, difícilmente sabremos cómo hacerlo. El problema es que, como vamos a tener que expresar con sumo detalle cómo hacerlo, también tenemos que saber con sumo detalle qué queremos hacer. A realizar este paso se le

denomina **especificar** el problema.

2 **Diseñar un plan**. Una vez que sabemos lo que queremos resolver necesitamos un plan. Este plan debería decirnos qué tenemos que hacer para resolver el problema.

3 **Implementar el plan**. Tener un plan no basta. Hay que llevarlo a cabo. Para llevarlo a cabo hay que escribirlo en un lenguaje que entienda el ordenador.

4 **Probar el plan**. Una vez implementado, hay que probar el plan. En la mayoría de los casos nos habremos equivocado o habremos ignorado una parte importante del problema que queríamos resolver. Sólo cuando probamos nuestra implementación y vemos lo que hace, podemos ver si realmente está resolviendo el problema (y si lo está haciendo de la forma que queríamos al implementar el plan).

5 **Depurar el plan**. Si nos hemos equivocado, lo que suele ser habitual, lo primero que necesitamos saber es en qué nos hemos equivocado. Posiblemente tengamos que probar el plan varias veces para tratar de ver cuál es el problema. Una vez localizado éste, tenemos que volver al punto en el que nos hemos equivocado (cualquiera de los mencionados antes).

Veamos en qué consiste cada uno de estos pasos. Pero antes, un aviso importante: **estos pasos no son estancos**, no están aislados. No se hace uno por completo y luego se pasa al siguiente. Si queremos tener éxito, tenemos que abordar el problema por partes y efectuar todos estos pasos para cada pequeña parte del problema original. De otro modo, no obtendremos nada útil, por muchos pasos que sigamos.

Esto último es tan importante que es la diferencia entre realizar programas que funcionan y construir auténticas atrocidades. **Las cosas se hacen poco a poco y por pasos, haciéndolo mal la primera vez y mejorando el resultado hasta que sea satisfactorio.**

Dijimos que programar es como cocinar y que un programa es como una receta de cocina. Piensa cómo llegan los grandes Chefs a conseguir sus recetas. Ninguno de ellos se sienta a trazar un plan maestro extremadamente detallado sin haber siquiera probado qué tal sabe una salsa. Tal vez, el chef se ocupe primero de hacer el sofrito, ignorando mientras tanto el resto de la receta. Quizás, se ocupe después de cómo asar la carne y de qué pieza asar (tal vez ignorando el sofrito). En ambos casos hará muchos de ellos hasta quedar contento con el resultado. Probará añadiendo una especia, quitando otra, etc. Programar es así.

Se programa **refinando progresivamente** el programa inicial, que posiblemente no cumple prácticamente nada de lo que tiene que cumplir para solucionar el problema, pero que al menos hace algo. En cada paso se produce un prototipo del programa final. En general, el programa se acaba de escribir cuando el prototipo cumple con toda la especificación y se dice basta en este proceso de mejora a fuerza de repetir los pasos mencionados anteriormente.

## 1.3. Refinamiento del programa para programar

Con lo dicho antes puede tenerse una idea de qué proceso hay que seguir para construir programas. Pero es mejor refinar nuestra descripción de este proceso, para que se entienda mejor y para conseguir el vocabulario necesario para entenderse con otros programadores. ¿Se ve cómo estamos refinando nuestro programa para programar? Igual hay que hacer con los programas.

Empecemos por definir el problema. ¿Especificación? ¿Qué especificación? Aunque en muchos textos se sugiere construir laboriosas y detalladas listas de requisitos que debe cumplir el software a construir, o seguir determinado esquema formal, nada de esto suele funcionar bien para la resolución de problemas abordables por programas que no sean grandes proyectos colectivos de software. Incluso en ese caso, es mucho mejor una descripción precisa aunque sea más informal que una descripción formal y farragosa que puede contener aún más errores que el programa que intenta especificar.

Para programar individualmente, o empleando equipos pequeños, es suficiente tener claro cuál es el problema que se quiere resolver. Puede ayudar escribir una descripción informal del problema, para poder acudir a ella ante cualquier duda. Es bueno incluir todo aquello que

queramos que el programa pueda hacer y tal vez mencionar todo lo que no necesitamos que haga. Por lo demás nos podemos olvidar de este punto.

Lo que importa en realidad para poder trabajar junto con otras personas son los **interfaces**. Esto es, qué tiene que ofrecerle nuestra parte del programa a los demás y qué necesita nuestra parte del programa de los demás. Si esto se hace bien, el resto viene solo. En otro caso, es mejor cambiar de trabajo (aunque sea temporalmente).

Una vez especificado el problema necesitamos un plan detallado para resolverlo. Este plan es el **algoritmo** para resolver el problema. Un algoritmo es una secuencia detallada de acciones que define cómo se calcula la solución del problema. Por poner un ejemplo, un algoritmo para freír un huevo podría ser el que sigue:

```
Tomar sarten
Si falta aceite entonces
       Tomar aceite y
       Poner aceite en sarten
Y despues seguir con...
Poner sarten en fogon
Encender fogon
Mientras aceite no caliente
       no hacer nada
Y despues seguir con...
Tomar huevo
Partir huevo
Echar interior del huevo a sarten
Tirar cascara
Mientras huevo crudo
       mover sarten (para que no se pegue)
Y despues seguir con...
Retirar interior del huevo de sarten
Tomar plato
Poner interior del huevo en plato
Apagar fogon
```

A este lenguaje utilizado para redactar informalmente un algoritmo se le suele denominar **pseudocódigo**, dado que en realidad este texto intenta ser una especie de código para un programa. Pero no es un programa: no es algo que podamos darle a un ordenador para que siga sus indicaciones, esto es, para que lo ejecute. Se puede decir que el pseudocódigo es un lenguaje intermedio entre el lenguaje natural (el humano) y el tipo de lenguajes en los que se escriben los programas. Luego volveremos sobre este punto.

En realidad un ordenador no puede ejecutar los algoritmos tal y como los puede escribir un humano. Es preciso escribir un programa (un texto, guardado en un fichero) empleando un lenguaje con una sintaxis muy estricta que luego puede traducirse automáticamente y sin intervención humana al lenguaje que en realidad habla el ordenador (binario, como ya dijimos).

Al lenguaje que se usa para escribir un programa se le denomina, sorprendentemente, **lenguaje de programación**. El lenguaje que de verdad entiende el ordenador se denomina **código máquina** y es un lenguaje numérico. Un ejemplo de un programa escrito en un lenguaje de programación podría ser el siguiente:

```
procedure freirhuevo()
{
       tomar(sarten);
       if (cantidad(aceite) < Minima) {
              tomar(aceite);
              poneren(sarten, aceite);
       }
       poneren(fogon, sarten);
       encender(fogon);
       while (not estacaliente(aceite)) {
              nohacernada();
       }
       tomar(huevo);
       partir(huevo, cascara, interior);
       poneren(sarten, interior);
       eliminar(cascara);
       while (estacrudo(huevo)) {
              mover(sarten);/* para que no se pegue */
       }
       tomar(plato);
       quitarde(sarten, interior);
       poneren(plato, interior);
       apagar(fogon);
}
```

Al texto de un programa (como por ejemplo este que hemos visto) se le denomina **código fuente** del programa. Programar es en realidad escribir código fuente para un programa que resuelve un problema. A cada construcción (o frase) escrita en un lenguaje de programación se le denomina **sentencia**. Por ejemplo,

```
eliminar(cascara);
```

es una sentencia.

Para introducir el programa en el ordenador lo escribimos en un fichero empleando un **editor**. Un editor es un programa que permite editar texto y guardarlo en un fichero. Como puede verse, todo son programas en este mundo. Es importante utilizar un editor de texto (como *TextEdit*, *Gedit*, o *SciTE*) y no un procesador de textos (un programa que sirve para escribir documentos, como *OpenOffice* o *Word*) puesto que estos últimos están más preocupados por el aspecto del texto que por el texto en sí. Si utilizamos un procesador de textos para escribir el programa, el fichero contendrá muchas más cosas además del texto (negritas, estilo, etc.), y no podremos traducir el texto a código máquina, puesto que el programa que hace la traducción no lo entenderá. El texto del programa debe ser lo que se denomina *texto plano*.

A la acción de escribir el texto o código de un programa se le suele denominar también **codificar** el algoritmo o **implementar** (o realizar) el algoritmo.

Una vez codificado, tenemos que traducir el texto a código máquina. Para esto empleamos un programa que realiza la traducción. Este programa se denomina **compilador**; se dice que compila el lenguaje de programación en que está escrito el código fuente. Por tanto, **compilar** es traducir el programa al lenguaje del ordenador empleando otro programa para ello (este último ya disponible en el lenguaje del ordenador, claro).

Un compilador toma un fichero como datos de entrada, escrito en el lenguaje que sabe compilar. A este fichero se le denomina **fichero fuente** del programa. Como resultado de su ejecución, el compilador genera otro fichero ya en el lenguaje del ordenador denominado **fichero objeto**. El fichero objeto ya está redactado en binario, pero no tiene todo el programa. Tan sólo tiene las partes del programa que hemos escrito en el fichero fuente. Normalmente hay muchos trozos del programa que tomamos prestados del lenguaje de programación que utilizamos y que se encuentran en otros ficheros objeto que ya están instalados en el ordenador, o que hemos

compilado anteriormente.

Piensa que, en el caso de las recetas de cocina, nuestro chef puede tener un pinche que sólo habla chino mandarín. Para hacer la receta, el chef, que es de Cádiz, escribe en español diferentes textos (cómo hacer el sofrito, cómo el asado, etc.). Cada texto se manda traducir por separado a chino mandarín (obteniendo el objeto del sofrito, del asado, etc.). Luego hay que reunir todos los objetos en una única receta en chino mandarín y dársela al pinche.

Una vez tenemos nuestros ficheros objeto, y los que hemos tomado prestados, otro programa denominado **enlazador** se ocupa de juntar o enlazar todos esos ficheros en un sólo binario que pueda ejecutarse. De hecho, suele haber muchos ficheros objeto preparados para que los usemos en nuestros programas. A estos ficheros se les llama **librerías** (o bibliotecas) del sistema. Las librerías son ficheros que contienen código ya compilado y listo para pasar al enlazador. Todo este proceso está representado esquemáticamente en la figura 1.2.

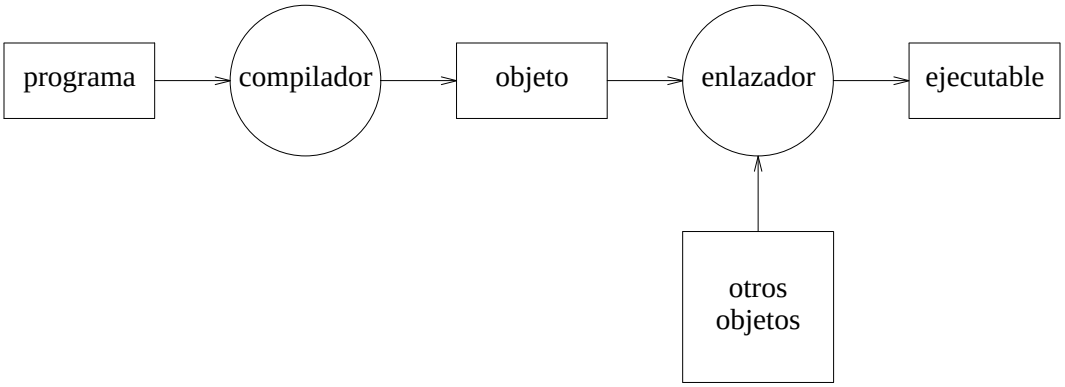



**Figura 1.2:** *El compilador genera ficheros objeto que se enlazan y forman un ejecutable.*

Un fichero ejecutable puede tener el aspecto que sigue, si utilizamos base 16 para escribir los números que lo forman (llamada **hexadecimal** habitualmente):

```
0000000  4e6f726d 616c6d65 6e746520 73652075
0000010  74696c69 7a612065 6c207465 636c6164
0000020  6f20792f 6f20656c 20726174 c3b36e20
0000030  70617261 20696e64 69636172 6c652061
0000040  6c0a7369 7374656d 61207175 6520676f
 ...
```

Claramente no queremos escribir esto directamente. El ejecutable contiene **instrucciones**, que son números de una longitud determinada que corresponden a órdenes concretas para la CPU. A estos números también se les conoce como **palabras**. El ejemplo anterior muestra cuatro palabras en cada línea. Los datos que maneja el programa en la memoria del ordenador tienen el mismo aspecto. Son palabras, esto es, números. Por eso se suele denominar binarios a los ficheros objeto y a los ficheros ejecutables, por que contienen números en base 2 (en binario) que es el lenguaje que en realidad entiende el ordenador.

Una vez tenemos un ejecutable, lo primero es ejecutar el programa. Normalmente se utiliza el teclado y/o el ratón para indicarle al sistema que gobierna el ordenador (al sistema operativo) que ejecute el programa. La ejecución de un programa consiste en la **carga** del mismo en la memoria del ordenador y en conseguir que la CPU comience a ejecutar las instrucciones que componen el programa (almacenadas ya en la memoria).

En general, todo programa parte de unos datos de entrada (normalmente un fichero normal o uno especial, como puede ser el teclado del ordenador) y durante su ejecución genera unos datos de salida o resultados (normalmente otro fichero, que puede ser normal o uno especial, como puede ser la pantalla del ordenador). En cualquier caso, a los datos los denominamos **entrada** del

programa y a los resultados producidos los denominamos **salida** del programa. El esquema lo podemos ver en la figura 1.3.

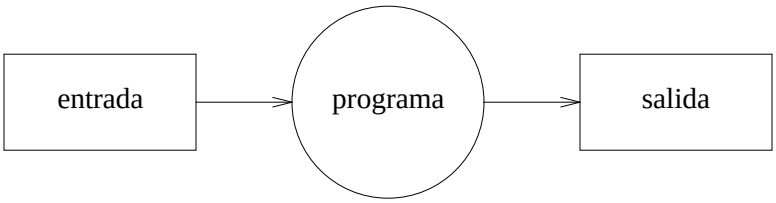



**Figura 1.3:** *Un programa toma datos de la entrada y tras procesarlos produce resultados en la salida.*

Las primeras veces que ejecutamos el programa estamos interesados en **probarlo** (comprobar que el resultado es el esperado). En muchos casos cometeremos errores al escribir un programa. Algunos de ellos son errores sintácticos que puede detectar el compilador. El compilador también hace todo lo posible para cubrirnos las espaldas y detectar errores en el código que nos llevarían a una ejecución incorrecta. Estos son **errores de compilación** que impiden al compilador generar un ejecutable. Si cometemos este tipo de errores deberemos arreglar el programa y volver a intentar su compilación. Pero ten esto en cuenta: un programa puede compilar y no ser correcto. El compilador hace todo lo posible, pero no puede hacer magia.

En otros casos, un error en el programa consiste en ejecutar instrucciones erróneas (por ejemplo, dividir por cero) que provocarán que el programa tenga problemas en su ejecución. Naturalmente el programa hace lo que se le dice que haga, pero podemos no haber tenido en cuenta ciertos casos que conducen a un error. Estos son **errores de ejecución** que provocan que el programa deje de ejecutar cuando se producen, dado que el ordenador no sabría qué hacer a continuación del error.

Además tenemos los llamados **errores lógicos**, que consisten en que el programa ejecuta correctamente pero no produce los resultados que esperamos (a lo mejor hemos puesto el huevo primero y la sartén después, y aunque esperábamos comer, nos toca limpiar el desastre en ayunas).

El propósito de probar el programa es intentar descubrir nosotros todos los errores posibles antes de que los sufran otros (y causen un perjuicio mayor). Si un programa no se prueba intentando romperlo por todos los medios posibles, lo más seguro es que el programa no funcione correctamente. Por cierto, sea cual sea el tipo de error, se le suele llamar **bug**. Siempre que alguien habla de un ‘‘bug’’ en un programa se refiere a un error en el mismo.

En cualquier caso, ante un error, es preciso ver a qué se debe dicho error y luego arreglarlo en el programa. Las personas que no saben programar tienden a intentar ocultar o arreglar el error antes siquiera de saber a qué se debe. Esto es la mejor receta para el desastre. Si no te quieres pasar horas y horas persiguiendo errores lo mejor es que no lo hagas.

Una vez se sabe a qué se debe el error se puede cambiar el programa para solucionarlo. Y por supuesto, una vez arreglado, hay que ver si realmente está arreglado ejecutando de nuevo el programa para probarlo de nuevo. A esta fase del desarrollo de programas se la denomina fase de pruebas. Es importante comprender la importancia de esta fase y entender que forma parte del desarrollo del programa.

Pero recuerda que una vez has probado el programa seguro que has aprendido algo más sobre el problema y posiblemente quieras cambiar la especificación del mismo. Esto hace que de nuevo vuelvas a empezar a redefinir el problema, rehacer un poco el diseño del algoritmo, cambiar la implementación para que corresponda al nuevo plan y probarla de nuevo. No hay otra forma de hacerlo.

## 1.4. Algoritmos

Normalmente un algoritmo (y por tanto un programa) suele detallarse empleando tres tipos de construcciones: secuencias de acciones, selecciones de acciones e iteraciones de acciones. La idea es emplear sólo estas tres construcciones para detallar cualquier algoritmo. Si se hace así, resultará fácil realizar el programa correspondiente y evitar parte de los errores.

Una **secuencia** es simplemente una lista de acciones que han de efectuarse una tras otra. Por ejemplo, en nuestro algoritmo para freír un huevo,

```
Tomar huevo
Partir huevo
Echar interior del huevo a sarten
Tirar cascara
```

es una secuencia. Hasta que no se termina una acción no comienza la siguiente, por lo que la ejecución de las mismas se produce según se muestra en la figura 1.4.

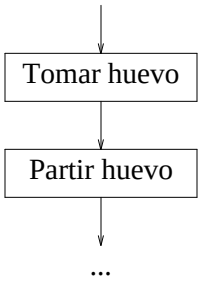



**Figura 1.4:** *Una secuencia ejecuta acciones una tras otra hasta concluir.*

Aquí lo importante es que **sistemáticamente se ejecuta una acción tras otra**. Siempre se comienza la ejecución por el principio (la parte superior de la figura) y se termina por el final (la parte inferior de la figura). En ningún momento se comienza directamente por una acción intermedia ni se salta desde una acción intermedia en la secuencia hasta alguna otra acción que no sea la siguiente.

La importancia de hacer esto así es que de no hacerlo resultaría tremendamente difícil comprender lo que realmente hace el algoritmo y sería muy fácil cometer errores. Errores que luego tendríamos que depurar (lo cual es justo lo que nunca queremos hacer, dado que depurar es una tarea dolorosa).

A este conjunto de acciones que comienzan a ejecutar en un único punto (arriba) y terminan en otro único punto (abajo) lo denominamos **bloque** de sentencias.

Otro elemento que utilizamos para construir algoritmos es la **selección**. Una selección es una construcción que efectúa una acción u otra dependiendo de que una condición se cumpla o no se cumpla. Por ejemplo, en nuestro algoritmo para freír un huevo,

```
Si falta aceite entonces
       Tomar aceite
       Poner aceite
Y despues seguir con...
```

es una selección. Aquí el orden de ejecución de acciones sería como indica la figura 1.5.

La condición es una pregunta que debe poder responderse con un sí o un no, esto es, debe ser una proposición verdadera o falsa. De este modo, esta construcción ejecuta una de las dos ramas. Por ejemplo, en la figura, o se ejecutan las acciones de la izquierda o se ejecuta la acción de la derecha. A cada una de estas partes (izquierda o derecha) se la denomina **rama** de la selección. Así, tenemos una rama para el caso en que la condición es cierta y otra para el caso en

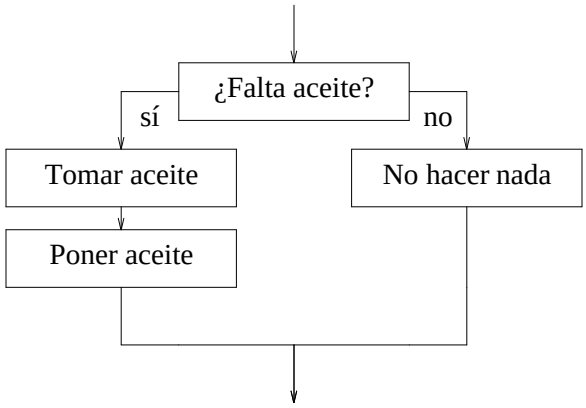



**Figura 1.5:** *Una selección ejecuta una u otra acción según una condición sea cierta o falsa.*

que la condición es falsa.

Igual que sucede con la secuencia, la ejecución siempre comienza por el principio (la parte superior de la figura) y termina justo detrás de la selección (la parte inferior en la figura). No es posible comenzar a ejecutar arbitrariamente. Tanto si se cumple la condición como si no se cumple (en cualquiera de las dos alternativas) podemos ejecutar una o más acciones (o quizá ninguna) en la rama correspondiente.

Podemos utilizar una selección en un algoritmo en cualquier sitio donde podamos utilizar una sentencia (dado que el flujo u orden en que se ejecutan las cosas empieza siempre por arriba y termina siempre por abajo). Además, como se ha podido ver, en cada rama podemos tener un bloque de sentencias (y no una única sentencia). Estos bloques pueden a su vez incluir selecciones, naturalmente.

Sólo hay otra construcción disponible para escribir algoritmos y programas: la **iteración**. Una iteración es una construcción que repite un bloque (una secuencia, una selección o una iteración) cierto número de veces. Por ejemplo, nuestro algoritmo culinario incluye esta iteración:

```
Mientras huevo crudo
      mover sarten
Y despues seguir con...
```

El orden de ejecución de esta construcción puede verse en la figura 1.6.

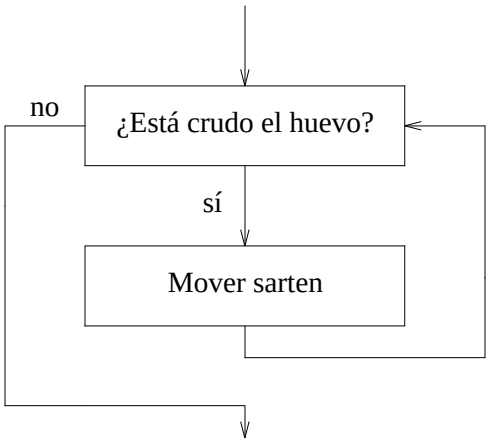



**Figura 1.6:** *Una iteración repite algunas sentencias mientras sea preciso.*

Como puede verse, también se comienza a ejecutar por un único punto al principio y se termina la ejecución en un único punto al final. En todos los casos (secuencia, selección o iteración) lo hacemos así. Esto hace que el algoritmo sea **estructurado** y sea posible seguir con facilidad la secuencia de ejecución de acciones. Aún mas, hace que sea posible cambiar con facilidad el algoritmo. Cuando hagamos un cambio podemos pensar sólo en la parte que cambiamos, dado que no es posible saltar arbitrariamente hacia el interior de la parte que cambiamos. Siempre se comienza por un punto, se hace algo y se termina en otro punto.

Cuando ejecutemos el programa que corresponde al algoritmo de interés, tendremos un orden de ejecución de acciones que podemos imaginar como una mano con un dedo índice que señala a la acción en curso y que avanza conforme continúa la ejecución del programa. A esto lo denominamos **flujo de control** del programa. El único propósito de la CPU es en realidad dar vida al flujo de control de un programa. Naturalmente no hay manos ni dedos en el ordenador. En su lugar, la CPU (la ‘‘mano’’) mantiene la cuenta de qué instrucción es la siguiente a ejecutar mediante el llamado **contador de programa** (el ‘‘dedo’’). Pero podemos olvidarnos de esto por el momento.

De igual modo que estructuramos el flujo de control empleando secuencias, selecciones e iteraciones, también es posible agrupar y estructurar los datos tal y como veremos en el resto del curso. Citando el título de un excelente libro de programación:

**Algoritmos + Estructuras de datos = Programas**

El mencionado libro [1] es un muy buen libro para leer una vez completado este curso.

## 1.5. Programas en Picky

En este curso de programación vamos a usar un lenguaje creado específicamente para él, llamado Picky. Picky es un lenguaje de programación que puede utilizarse para escribir programas de ordenador, los cuales sirven por supuesto para manipular datos en el ordenador.

Picky, como cualquier lenguaje de programación, manipula entidades **abstractas**, esto es, que no existen en realidad más que en el lenguaje. Quizá sorprendentemente, a estas entidades se las denomina datos abstractos.

Podemos tener lenguajes más abstractos (más cercanos a nosotros) y lenguajes más cercanos al ordenador, o menos abstractos. Por eso hablamos del **nivel de abstracción** del lenguaje. En Picky el nivel de abstracción del lenguaje está lejos del nivel empleado por el ordenador. Por eso decimos que Picky es un **lenguaje de alto nivel** (de abstracción, se entiende).

Esto es fácil de entender. Picky habla de conceptos (abstracciones) y el ordenador sólo entiende de operaciones elementales y números (elementos concretos y no muy abstractos). Por ejemplo, Picky puede hablar de símbolos que representan días de la semana aunque el ordenador sólo manipule números enteros (que pueden utilizarse para identificar días de la semana). Sucede lo mismo en cualquier otro lenguaje de este nivel de abstracción.

Antes de seguir... ¡Que no cunda el pánico! El propósito de este epígrafe es tomar contacto con el lenguaje. No hay que preocuparse de entender el código (los programas) que aparece después. Tan sólo hay que empezar a ver el paisaje para que luego las cosas nos resulten familiares. Dicho de otro modo, vamos a hacer una visita turística a un programa escrito en Picky.

Un programa Picky tal y como los que vamos a realizar durante este curso presenta una estructura muy definida y emplea una sintaxis muy rígida. Lo que sigue es un ejemplo de programa. Lo vamos a utilizar para hacer una disección (como si de una rana se tratase) para que se vean las partes que tiene. Pero primero el código:

**programa.p**

```
1     /*
2      *    Programa de bienvenida en Picky.
3      *    Autor: aturing
4      */

6     /*
7      *    Nombre del programa.
8      */
9     program programa;

11    /*
12     *    Tipos de datos
13     */
14    types:
15         /* Dias de la semana */
16         TipoDiaSem = (Lun, Mar, Mier, Jue, Vie, Sab, Dom);
17
18    /*
19     *    Constantes
20     */
21    consts:
22         Pi = 3.1415926;
23         RadioPrueba = 2.0;

25    /*
26     *    Funciones y procedimientos.
27     */

29    function longitudcirculo(r: float): float
30    {
31         return 2.0 * Pi * r;
32    }

34    /*
35     *    Programa Principal.
36     */
37    procedure main()
38    {
39         writeln("Hola que tal.");
40         writeln(longitudcirculo(RadioPrueba));
41    }
```

Lo primero que hay que decir es que este programa debe estar guardado en un fichero en el ordenador. De otro modo no sería útil en absoluto (salvo que tengamos que empapelar algo). En este libro escribimos el nombre de cada fichero en un pequeño rectángulo en negrita al principio de cada programa. Por ejemplo, el programa que hemos visto se guardaría en un fichero llamado `programa.p`.

Nótese que los números de línea mostrados a la izquierda de cada línea no son parte del programa (no hay que escribirlos). Los hemos incluido al presentar los programas para que sea más fácil referirse a una línea o líneas concretas.

Este fichero fuente se utilizará como entrada para un compilador de Picky, con el propósito de obtener un ejecutable como salida del compilador, tras la fase de enlazado, y poder ejecutar en el ordenador nuestro programa. Recuerda, todos los programas son lo mismo: utilizan una entrada, hacen algo y producen una salida.

Todo el texto mostrado constituye el **código fuente** del programa. Como puede verse, es texto escrito directamente sin emplear ni negrita, ni itálica, ni ningún otro elemento de estilo. Hay que tener en cuenta que lo único que le importa al compilador de Picky es el texto que hemos escrito y no cómo lo hemos escrito. Por eso dijimos que hay que emplear para escribir programas editores de texto y no programas como *Word*, que están en realidad más preocupados por cómo queda el texto que por otra cosa (y por consiguiente incluyen en el fichero información sobre cómo hay que presentar el texto; lo que confundiría al compilador).

Comencemos con la anatomía del programa. Llama la atención que hay líneas en el programa que están englobadas entre los símbolos ‘‘`/*`’’ y ‘‘`*/`’’. Estas líneas son **comentarios** y son líneas que el compilador ignora por completo. Esto es, todo el texto desde un ‘‘`/*`’’ hasta el ‘‘`*/`’’ (estos incluidos) es texto que podríamos borrar del programa si quisiéramos, y el programa seguiría siendo el mismo. No es necesario que esté en distintas líneas. Por ejemplo:

```
/* esto es un comentario */
```

Los comentarios están pensados para los humanos que leen el programa (con frecuencia los mismos humanos que han escrito el programa).

Es importante que un programa sea fácilmente **legible** por humanos. Esto es aún más importante que otros criterios tales como la eficiencia a la hora de utilizar los recursos del ordenador; los comentarios ayudan. Por ejemplo, las primeas líneas del programa anterior son un comentario:

```
1    /*
2     *    Programa de bienvenida en Picky.
3     *    Autor: aturing
4     */
```

Este comentario ayuda a que veamos una descripción del programa. A lo mejor, con leer ese comentario nos basta para ver qué hace, y nos ahorramos leer el programa entero.

Esta es la sintaxis que Picky impone para los comentarios. Otros lenguajes lo hacen de otro modo.

Otra cosa que podemos ver mirando el programa anterior es que un programa tiene varias secciones bien definidas, que habitualmente vamos a tener en todos los programas. En este curso nuestros programas tendrán siempre estas secciones en el orden que mostramos, aunque algunas de ellas podrán omitirse si están vacías en un programa dado. En nuestro programa hemos incluido comentarios al principio de cada sección para explicar el propósito de la misma. Vamos a verlas en orden.

La primera sección ya la conocemos. Es un comentario mostrando el propósito del programa y el autor del mismo. Por ejemplo:

```
/*
 *     Programa de ejemplo en Picky.
 *     Autor: Sheldom
 */
```

Algo informal y breve que diga qué es esta cosa que estamos mirando es en realidad el comentario más útil que podríamos hacer.

A continuación se especifica el nombre del programa.

```
7    /*
8     *    Nombre del programa.
9     */
10   program programa;
```

Aquí, la palabra *program* es una palabra reservada que se utiliza para indicar que *programa* es a partir de este momento el nombre (o identificador) del programa que estamos escribiendo. Dado que el propósito de este programa es mostrar un programa en Picky, *programa* es un buen

nombre. Pensemos que este programa no tiene otra utilidad que la de ser un programa. Por lo demás, es un programa que no sirve para gran cosa. Si hiciésemos un programa para escribir la tabla de multiplicar del 7, un buen nombre sería *tabladel7*. En ese caso, llamarlo *programa* sería una pésima elección. ¡Por supuesto que es un programa! El nombre debería decir algo que no sepamos sobre el programa. En realidad, debería decirlo todo en una sola palabra.

Normalmente, el nombre del programa se hace coincidir con el nombre del fichero que contiene el programa. Ahora bien, el nombre del fichero fuente de Picky ha de terminar en ‘‘`.p`’’. Por ejemplo, nuestro programa de ejemplo para esta disección debería estar en un fichero llamado ‘‘`programa.p`’’.

Como hemos visto, a lo largo del programa emplearemos palabras que tienen un significado especial. Aquí, la palabra *program* es una de esas palabras; decimos que es una **palabra reservada** o *palabra clave* del lenguaje (en inglés, *keyword*). Esto quiere decir que es una palabra que forma parte del lenguaje y se utiliza para un propósito muy concreto. No podemos usar la palabra *program* para ningún otro propósito que para especificar el nombre del programa. Lo mismo pasa con las demás palabras reservadas.

En un programa también utilizamos otras palabras para referirnos a elementos en el lenguaje o a elementos en el mundo (según lo ve el programa, claro). Las llamamos identificadores. Por ejemplo, *programa* es un **identificador**. Los identificadores se llaman así porque identifican elementos en nuestro programa. En general, tendremos algunos identificadores ya definidos que podremos utilizar y también podremos definir otros nosotros. En este caso, como es el nombre de nuestro programa, lo hemos puesto nosotros.

Los identificadores son palabras que deben comenzar por una letra y sólo pueden utilizar letras o números (pero teniendo en cuenta que a ‘‘_’’ se le considera una letra). No pueden comenzar con otros símbolos tales como puntos, comas, etc. Piensa que el compilador que debe traducir este lenguaje a binario es un programa, y por lo tanto, no es muy listo. Si no le ayudásemos empleando una sintaxis y normas férreas, no sería posible implementar un compilador para el lenguaje.

Ahora que los mencionamos, a los símbolos que utilizamos para escribir (letras, números, signos de puntuación) los denominamos **caracteres**. Incluso un espacio en blanco es un carácter.

Los siguientes nombres son identificadores correctos en Picky:

```
Imprimir_Linea
Put0
getThisOrThat
X32__
```

Pero estos otros **no** lo son:

```
0x10
Fecha del mes
0punto
```

Si no ves por qué, lee otra vez las normas para escribir identificadores que acabamos de exponer y recuerda que el espacio en blanco es un carácter como otro cualquiera. El segundo identificador de este ejemplo no es válido puesto que utiliza caracteres blancos (esto es, espacios en blanco) como parte del nombre.

Por cierto, Picky distingue entre mayúsculas y minúsculas. Los ordenadores, en general, hacen esta diferenciación. Por lo tanto, *Programa*, *programa*, *PROGRAMA* y *ProGraMa* son diferentes identificadores. Deberás recordar que esto es así en Picky, pero hemos de decir que en otros lenguajes no se distingue entre mayúsculas y minúsculas, y todos los ejemplos anteriores serían el mismo identificador.

Tras el nombre del programa, tenemos otra sección dedicada a definir nuevos elementos en el mundo donde vive nuestro programa. Esta sección se denomina sección de definición de tipos de datos (que ya veremos lo que son). Por ejemplo, en nuestro programa estamos definiendo los

días de la semana para que el programa pueda manipular días.

```
13    /*
14     *    Tipos de datos
15     */
16    types:
17         /* Dias de la semana */
18         TipoDiaSem = (Lun, Mar, Mier, Jue, Vie, Sab, Dom);
```

Suele ser útil indicar, en un comentario antes de cada definición, el propósito de la misma. Aunque suele ser aún mejor que la definición resulte obvia. Dados los nombres empleados en esta definición, no es necesario usar un comentario. Con ver las siguientes dos líneas, se puede entender perfectamente que se trata de los días de la semana:

```
types:
       TipoDiaSem = (Lun, Mar, Mier, Jue, Vie, Sab, Dom);
```

En general, el código es más fácil de leer si no tenemos comentarios inútiles. Pero atención, esto no quiere decir que no tenga que haber comentarios. Esto quiere decir que hay que prescindir de los comentarios que no dan información útil, y que el código debe ser lo más autoexplicativo posible. A medida que avancemos con el curso, iremos viendo más ejemplos.

La palabra *types* es una palabra reservada del lenguaje que se utiliza (seguida por dos puntos) para marcar el inicio de la sección de definición de tipos de datos, y no podemos emplear ninguna otra en su lugar. En cambio, *TipoDiaSem*, *Lun*, *Mar*, etc. son identificadores, y podríamos haber empleado otros, ya que los identificadores los elige el programador.

La siguiente sección es la de definición de constantes. Como indica su nombre, son valores que nunca cambian (ya veremos para qué sirven en los siguientes capítulos). En este caso se usa la palabra reservada *consts* seguida de dos puntos para indicar el comienzo de la sección.

```
20    /*
21     *    Constantes
22     */
23    consts:
24         Pi = 3.1415926;
25         RadioPrueba = 2.0;
```

Aquí, de nuevo, vemos algunos identificadores como *Pi* y *RadioPrueba*, cuyos nombres dan una noción bastante clara de su propósito.

En estas líneas, podemos ver además que en un programa hay símbolos que se representan a sí mismos. Por ejemplo, 3.1415926 es un número real concreto, y *Mar* es un día de la semana concreto. Estos símbolos se denominan **literales** y se utilizan para referirse a ciertos elementos literalmente. Por ejemplo, quizá resulte sorprendente que el literal 3.1415926 se refiere al número 3.1415926 de forma literal. Por eso lo denominamos literal, literalmente.

Como ya se ha comentado anteriormente, Picky distingue mayúsculas de minúsculas. Se suelen respetar normas respecto a cuándo emplear mayúsculas y cuándo minúsculas a la hora de escribir los identificadores y palabras reservadas en un programa. Hacerlo así tiene la utilidad de que basta ver un identificador escrito para saber no sólo a qué objeto se refiere, sino también de qué tipo de objeto se trata.

Las palabras reservadas se tienen que escribir siempre en minúscula. Los nombres de constantes los escribiremos siempre comenzando por mayúscula (con el resto en minúsculas). Ten en cuenta que al elegir nosotros mismos su identificador, podríamos escribirlo en minúsculas y para el compilador no habría ningún problema. Pero las vamos a escribir comenzando en mayúsculas para que, de un solo vistazo, podamos diferenciarlas de una palabra reservada. Así pues, *Pi* parece ser una constante. Los nombres de tipos los capitalizamos igual que los de constantes. Por ejemplo, *TipoDiaSem* en nuestro caso. Eso sí, haremos siempre que el nombre empiece por *Tipo* (para saber que hablamos de un tipo; aunque ahora mismo no sepamos de lo que hablamos).

La siguiente sección en nuestro programa es la de funciones y procedimientos. Esta sección define pequeños programas auxiliares (o **subprogramas**) que sirven para calcular algo necesario para el programa que nos ocupa. En nuestro ejemplo, esta sección es como sigue:

```
27    /*
28     *    Funciones y procedimientos.
29     */

31    function longitudcirculo(r: float): float
32    {
33         return 2.0 * Pi * r;
34    }
```

Este fragmento de programa define la función *longitudcirculo*, similar a la función matemática empleada para la calcular la longitud correspondiente a un círculo de radio *r*. Es aconsejable incluir un breve comentario antes de la función indicando el propósito de la misma. Aunque, como dijimos, si los nombres se escogen adecuadamente, posiblemente el comentario resulte innecesario. Así, una función llamada *longitudcirculo* seguramente no haga otra cosa que calcular la longitud de un círculo (de hecho, ¡no debería hacer otra cosa!).

Esta función recibe un número real y devuelve un número real, similar a lo que sucede con una función matemática con domino en $\mathbb{R}$ e imagen en $\mathbb{R}$. El resultado para *longitudcirculo(r)* sería el valor de $2\pi r$. Luego volveremos sobre esto, aunque puede verse que

```
33              return 2.0 * Pi * r;
```

es la sentencia que indica cómo calcular el valor resultante de la función.

Es extremadamente importante probar todas las funciones y subprogramas antes de utilizarlos. De otro modo no podemos estar seguros de que funcionen bien. Es más, en otro caso podemos estar realmente seguros de que *no* funcionan bien.

Y por último, nos falta lo más importante: el programa principal. En nuestro caso es como sigue:

```
36    /*
37     *    Programa Principal.
38     */

40    procedure main()
41    {
42         writeln("Hola que tal.");
43         writeln(longitudcirculo(RadioPrueba));
44    }
```

Lo que hay entre ‘‘{’’ y ‘‘}’’ se conoce como el **cuerpo** del programa. A este programa lo llamamos **programa principal**, en consideración a que los subprogramas (funciones y procedimientos) que forman parte del fuente son en tambien programas. En cualquier caso, cuando ejecutemos el programa, éste empezará a realizar las acciones o sentencias indicadas en el cuerpo del programa principal. Eso sí, tras efectuar las definiciones encontradas antes.

En Picky, el programa principal siempre se llama *main*. Todo programa escrito en Picky tiene que tener un procedimiento llamado *main*.

Podemos ver que normalmente el programa principal incluye sentencias para efectuar pruebas (que luego desparecerán una vez el programa funcione) y sentencias para realizar las acciones que constituyen el algoritmo que queremos emplear. Como ya hemos visto antes, se usarán secuencias, selecciones e iteraciones para implementar el algoritmo.

En este programa principal utilizamos una sentencia que resultará muy útil a lo largo de todo el curso. La sentencia *writeln* escribe una línea en la pantalla con algo que le indicamos entre paréntesis. Hay otra sentencia, *write*, que también escribe, pero sin saltar a la siguiente línea. En

este ejemplo, el programa principal es una secuencia con dos sentencias: la primera escribe una línea en la pantalla saludando al usuario y la segunda escribe una línea en la pantalla con el resultado de calcular la longitud de un círculo.

Hemos podido ver a lo largo de todo este epígrafe que ciertas declaraciones y sentencias están escritas con sangrado, llamado también **tabulación**, de tal forma que están escritas más a la derecha que los elementos que las rodean. Esto se hace para indicar que éstas se consideran dentro de la estructura definida por dichos elementos.

Por ejemplo, las sentencias del programa principal están dentro del bloque de sentencias escrito entre llaves:

```
41    {
42          writeln("Hola que tal.");
43          writeln(longitudcirculo(RadioPrueba));
44    }
```

Por eso las sentencias en las líneas 42 y 43 están tabuladas a la derecha de ‘‘`{`’’ y ‘‘`}`’’ (debido a que dichas sentencias están dentro de ‘‘`{`’’ y ‘‘`}`’’ o dentro del cuerpo del programa). Están escritas comenzando por un tabulador (que se escribe pulsando la tecla del tabulador en el teclado, indicada con una flecha a la derecha en la parte izquierda del teclado). Así, en cuanto vemos el programa podemos saber a simple vista que ciertas cosas forman parte de otras. Esto es muy importante, puesto nos permite **olvidarnos** de todo lo que esté dentro de otra cosa (a no ser que esa cosa sea justo lo que nos interese).

Todas las definiciones o declaraciones están sangradas o tabuladas; lo mismo que sucede con el cuerpo del programa (las sentencias entre las llaves).

Un último recordatorio: los signos de puntuación son importantes en un programa. El compilador depende de los signos de puntuación para reconocer la sintaxis del programa. Por ahora diremos sólo dos cosas respecto a esto:

1 En Picky, todas las sentencias terminan con un ‘‘`;`’’. Esto no quiere decir que todas las líneas del programa terminen en un punto y coma. Quiere decir que las sentencias (y las declaraciones) han de hacerlo. Por ejemplo, *procedure* no tiene nunca un punto y coma detrás.

2 Los signos tales como paréntesis y comillas deben de ir por parejas y bien agrupados. Esta sentencia

```
writeln("Hola que tal.")
```

es incorrecta en Picky dado que le falta un ‘‘`;`’’ al final. Igualmente,

```
writeln "Hola que tal.")
```

es incorrecta dado que le falta un paréntesis. Del mismo modo,

```
(3 + ) (3 - 2 )
```

es una expresión incorrecta dado que el signo ‘‘`+`’’ requiere de un segundo valor para sumar, que no encuentra antes de cerrar la primera pareja de paréntesis. Todo esto resultará natural conforme leamos y escribamos programas y no merece la pena decir más por ahora.

## 1.6. ¡Hola π!

Para terminar este capítulo vamos a ver un programa en Picky denominado ‘‘Hola π’’. Este programa resultará útil para los problemas que siguen y para el próximo capítulo.

**holapi.p**

```
1     /*
2      *    Saludar al numero π.
3      */

5     program holapi;

7     consts:
8          Pi = 3.1415926;

10    procedure main()
11    {
12         write("Hola ");
13         write(Pi);
14         write("!");
15         writeeol();
16    }
```

Este programa define una constante llamada *Pi* en la línea 8 y luego las sentencias del programa principal se ocupan de escribir un mensaje de saludo para dicha constante. Por ahora obviaremos el resto de detalles de este programa.

Para compilar este programa nosotros daremos la orden ‘‘`pick`’’ al sistema operativo indicando el nombre del fichero que contiene el código fuente. Cuando damos una orden como esta al sistema operativo, estamos ejecutando un comando. En este caso, ‘‘ejecutamos el comando *pick*’’. Si el compilador no escribe ningún mensaje de error, entonces habrá sido capaz de traducir el código fuente y generar el programa. El nombre del fichero generado es, por omisión, `out.pam`. Para ejecutar el programa, basta con ejecutar el fichero binario generado. Por ejemplo:

```
; pick holapi.p
; out.pam
Hola 3.141593!
```

En adelante mostraremos la salida de todos nuestros programas de este modo. Lo que el programa ha escrito en este caso es

```
Hola 3.141593!
```

y el resto ha sido sencillamente lo que hemos tenido que escribir nosotros en nuestro sistema para ejecutar el programa (pero eso depende del ordenador que utilices).

En adelante utilizaremos siempre en nuestros ejemplos ‘‘`;`’’ como símbolo del sistema (o *prompt*) de la línea de comandos. Todo el texto que escribimos nosotros en el ordenador se presenta en texto *`ligeramente inclinado`* o italizado (cursiva). El texto escrito por el ordenador o por un programa se presenta siempre `sin inclinar`.

Puedes conseguir el software necesario para programar en Picky (el compilador) en

```
http://lsub.org/ls/picky.html
```

**Problemas**

1 Compila y ejecuta el programa ‘‘hola π’’ cuyo único propósito es escribir un saludo. Copia el código del programa o tómalo de la página de recursos de la asignatura.

2 Borra un punto y coma del programa. Compílalo y explica lo que pasa. Soluciona el problema.

3 Borra un paréntesis del programa. Compílalo y explica lo que pasa. Soluciona el problema.

4 Elimina la última línea del programa. Compílalo y explica lo que pasa. Soluciona el

problema.

5 Elimina la línea que contiene la palabra reservada *program*. Compílalo y explica lo que pasa. Soluciona el problema.

6 Escribe tú el programa desde el principio, compílalo y ejecútalo.

7 Cambia el programa para que salude al mundo entero y no sólo al número π, por ejemplo, imprimiendo `hola mundo!`.