Capítulo 1

PRINCIPIOS BÁSICOS DE LA PROGRAMACIÓN

En este capítulo veremos en qué consiste la tarea de programar, qué herramientas se necesitan para llevarla a cabo, qué herramientas hay disponibles y cuáles conviene elegir según el tipo de programación que se elija. Además, construiremos un pequeño programa para contar las palabras de una frase, el cual nos servirá para tener nuestra primera experiencia con el lenguaje Pascal.

¿Qué es programar?
(Conceptos básicos)

➔ La tarea de escribir un programa o programar consiste en escribir detallada y minuciosamente las **instrucciones** que debe seguir una computadora para realizar una tarea. Estas instrucciones deben escribirse en un lenguaje que la computadora pueda entender, ya sea en forma directa o luego de una traducción realizada por un **intérprete** o un **compilador**, como se explica más adelante en este capítulo.

El concepto más importante que hay que comprender antes de ponerse a programar es que la computadora carece totalmente de sentido común. Los programas no deben (idealmente) dejar nada librado al azar, deben cubrir todos los posibles casos que puedan suceder. Por ejemplo, imaginemos que mediante un programa queremos instruir una computadora para que maneje un auto. El programa podría ser más o menos así:

1. abrir la puerta del lado del conductor
2. entrar y sentarse frente al volante
3. introducir la llave de encendido
4. girar la llave de encendido presionando levemente el acelerador
5. presionar el embrague
6. usando la palanca de cambios, poner primera marcha
7. soltar el embrague gradualmente mientras se aumenta la presión sobre el acelerador, etc.

Estas instrucciones son, en apariencia, lo suficientemente detalladas como para que cualquiera, incluso una computadora, pueda seguirlas. Sin embargo, hay una gran cantidad de situaciones que este programa no prevé y que podrían derivar en problemas de distinta índole. Por ejemplo, ¿qué pasaría si frente al auto hubiera una pared? Un ser humano con el mínimo sentido común sabría que, en vez de poner primera marcha (como le indican las instrucciones), debería poner marcha atrás para arrancar. Pero la computadora hace exactamente lo que le indica el programa, con lo cual, por más que vea que hay una pared por delante, pondría primera y se chocaría tranquilamente con ella.

Es por eso que los programas deben prever cualquier eventualidad que pueda

ocurrir lidiando con ella debidamente. En el caso de nuestro programa para conducir un auto, deberíamos reemplazar la instrucción:

```
6. usando la palanca de cambios, poner primera marcha
```

por un conjunto de instrucciones que involucran el análisis de una condición:

```
6. si no hay obstáculos adelante:
   usando la palanca de cambios poner primera marcha
   si no:
   usando la palanca de cambios poner marcha atrás
```

De esta forma, le estamos dando al programa la capacidad de **decidir** si debe avanzar o retroceder el auto en función de la existencia de obstáculos delante del mismo. De paso, vamos captando la idea de lo que son las **estructuras de decisión**, que veremos más profundamente en el **Capítulo 4**.

Para que nuestro programa sea realmente "robusto" (se denomina así a un programa cuando tiene muy pocas probabilidades de fallar), tendríamos que evaluar muchas otras posibles condiciones, por ejemplo, que tampoco haya obstáculos detrás del auto, que el auto tenga nafta en el tanque, si fuera de noche, encender las luces, etc.

Probablemente, una de las tareas más difíciles de la programación sea prever todas las causas posibles que pueden provocar que un programa falle. Tanto es así que ningún programador en sus cabales es capaz de afirmar con total seguridad que un programa está ciento por ciento libre de errores.

PROGRAMACIÓN DE JUEGOS

En los albores de la programación de juegos para PC, el lenguaje ensamblador era el preferido por esta "raza" de programadores. Con el tiempo fueron apareciendo librerías de alto nivel para manejo de gráficos y animaciones, y hoy los programadores de juegos prefieren lenguajes estructurados u orientados a objetos, como el C o el C++, para hacer sus creaciones. El lenguaje C++ conserva todas las capacidades de su antecesor y le suma ventajas respecto del manejo de memoria.

LENGUAJES DE PROGRAMACIÓN: ¿POR QUÉ HAY TANTOS?

(PARADIGMAS, LENGUAJES Y FORMAS DE PROGRAMAR)

→ En párrafos anteriores vimos que los programas se escriben en lenguajes que pueden traducirse a una forma que la computadora pueda entender (como veremos más adelante, a esta forma se la llama **lenguaje de máquina**). Esto hizo que, a lo largo de la historia de la programación, se fueran creando distintos lenguajes para distintas necesidades, cada uno con su correspondiente "traductor" a lenguaje de máquina. Por lo general, los lenguajes de programación surgieron de centros de investigación en universidades o empresas, cada uno con el objetivo de cubrir alguna necesidad en particular.

Así nacieron, por ejemplo, el **FORTRAN** (por *Formula Translator*, o traductor de fórmulas), pensado especialmente para que los científicos de distintas disciplinas pudieran escribir programas para hacer cálculos de gran complejidad, y el **COBOL** (por *Common Business Oriented Language* o lenguaje común orientado a negocios), que fue pensado con el objetivo de escribir programas para administrar empresas.

LOS DISTINTOS PARADIGMAS Y SUS LENGUAJES

Con el tiempo, los investigadores en ciencias de la computación observaron que, más allá del propósito para el que fueron creados, los lenguajes podían diferenciarse por la forma de trabajo que presentan al programador, ofreciendo diversas formas de "ver" y "pensar" un programa antes de escribirlo.

Así comenzaron a surgir distintos **paradigmas de programación**, cada uno representado por una familia de lenguajes (**Figura 1**). Por ejemplo, a la forma tradicional de programar –que consiste en detallar la secuencia de pasos que debe seguir la computadora para realizar una tarea– se la denominó **programación imperativa**, porque el programa da órdenes que la computadora debe cumplir obedientemente.

LOS LENGUAJES .NET

El entorno .NET de Microsoft presenta un marco de trabajo (llamado Framework) en el que se pueden utilizar lenguajes muy diversos. Todos ellos tienen la particularidad de ser orientados a objetos y de compartir una misma interfaz mediante la cual interactúan con el sistema operativo.

Figura 1. *Los lenguajes más comunes –C, Pascal, Basic, FORTRAN, etc.– pertenecen al paradigma de programación imperativa; los dos primeros enfatizan el concepto de programación estructurada.*

Una rama dentro de la programación imperativa es la **programación estructurada**, cuyo propósito fundamental es construir programas claros, fáciles de entender y de mantener. Para ello se basa en el uso de módulos independientes (funciones y procedimientos) que separan claramente las diferentes tareas que

realiza un programa y en estructuras que conducen claramente el flujo de ejecución (más sobre esto en el **Capítulo 4**). Algunos lenguajes que enfatizan el concepto de programación estructurada son el C y el Pascal.

En contraposición a la programación imperativa se encuentra el paradigma de **programación declarativa**, que en lugar de dar una secuencia detallada de órdenes para que la computadora lleve a cabo, simplemente enuncia el resultado que se desea obtener y se deja que la computadora elija la forma más conveniente de obtenerlo. Este paradigma no es tan común como el de programación imperativa y se usa para aplicaciones que involucran inteligencia artificial, simulaciones científicas, etc. Dentro del paradigma de programación declarativa se encuentran los lenguajes de **programación lógica**, como el ProLog y el Lisp, los lenguajes de **programación funcional**, como Haskell, y otros lenguajes más específicos, como el SQL (*Structured Query Language*), cuyo propósito es el armado de consultas de bases de datos.

Otro paradigma importante derivado de la programación imperativa es el de la **programación orientada a objetos**. De gran popularidad en nuestros días, este paradigma permite pensar una aplicación como un conjunto de objetos autónomos que interactúan entre sí enviándose mensajes, con los cuales un objeto le "pide" a otro que realice una determinada tarea o le solicita determinados datos (**Figura 2**). La ventaja de esta forma de programación es que el desarrollo de las funciones está **encapsulado** dentro de cada objeto. En otras palabras, a un objeto no le preocupa cómo hace otro objeto para cumplir su función; le basta saber que, cuando se lo pida, hará su trabajo.

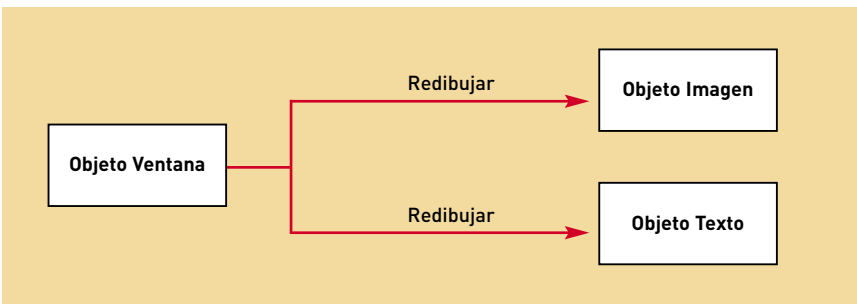


Figura 2. *En la programación orientada a objetos se facilita la modificación y el mantenimiento de los programas, ya que cualquier modificación interna a un objeto no afectará a los otros.*

¿QUÉ LENGUAJE CONVIENE ELEGIR?

La elección de un lenguaje de programación no es algo trivial. Antes que nada, debemos determinar qué clase de programas pensamos hacer. Si, por ejemplo, apuntamos a hacer programas que manejen bases de datos, nos convendrá dominar el lenguaje SQL. Pero el SQL no es muy adecuado para crear interfaces de usuario, por lo cual deberemos complementarlo con algún lenguaje que sí lo sea; por ejemplo, algún lenguaje que ofrezca un **entorno visual**, como Visual Basic o Delphi.

Los lenguajes puramente declarativos –el Haskell, el ProLog o el Lisp– son usados principalmente por investigadores y programadores muy vanguardistas en áreas como la inteligencia artificial.

Para tareas de programación más cotidianas –como podría ser escribir un programa para ordenar un conjunto de datos–, la mayoría de los programadores emplea lenguajes de propósito general propios de la programación estructurada, entre los cuales los más representativos son el Pascal y el C. Estos dos lenguajes son similares en su capacidad expresiva, pudiéndose utilizar cualquiera de los dos indistintamente para implementar cualquier programa estructurado. En particular, el Pascal es más fácil de aprender debido a que la mayoría de sus instrucciones son vocablos del idioma inglés (lamentablemente, no hay un Pascal en español), mientras que el C es más sintético, pues utiliza menos palabras y más símbolos para representar las instrucciones Tanto el Pascal como el C tienen versiones orientadas a objetos (en el caso del C se denomina C++), lo cual les permite mantenerse vigentes ante las "nuevas modas" en materia de programación, a diferencia de otros lenguajes, como por ejemplo, el FORTRAN o el COBOL, que han caído en desuso.

PSEUDOCÓDIGO

Existe un lenguaje de programación que, curiosamente, no se utiliza para crear programas ejecutables. Sirve solamente para que el programador exprese sus ideas de una forma clara y que estas ideas puedan comunicarse a otros programadores para que ellos las implementen en el lenguaje que prefieran.

INTERFAZ

Interfaz es todo aquello que media entre dos mundos diferentes creando la posibilidad de comunicación; en este caso, entre el hombre y la máquina. En la actualidad las interfaces están compuestas por ventanas, botones y barras, pero en el comienzo de la programación todo se realizaba por medio de comandos (o sea, escribiendo).

Este lenguaje del que hablamos es el **pseudocódigo**. A pesar de que no sirve para crear en forma directa programas ejecutables, es una herramienta de gran utilidad para programar "en papel"; es decir, para esbozar cómo un programa debe llevar a cabo su tarea.

Por ejemplo, el siguiente programa en pseudocódigo se utiliza para encontrar el mayor elemento dentro de una lista de datos:

```
Función BuscarMaximo(lista)
    Mayor = lista(1)
    Contador = 2
    Mientras Contador ≤ longitud(lista) hacer
        Si lista(Contador) > Mayor entonces
            Mayor = lista(Contador)
        Fin Si
        Contador = Contador + 1
    Fin Mientras
    Devolver Mayor
Fin Función
```

Dado que no se trata de un lenguaje "formal", nos da la libertad de escribirlo en el idioma de nuestra preferencia (en este caso, en español).

Otra gran ventaja del pseudocódigo es que nos permite hacer **refinamientos sucesivos** de un programa; esto quiere decir que nuestro programa, al principio, puede expresar las instrucciones de una forma muy general (por ejemplo, en vez de detallar todas las instrucciones para recorrer una lista de datos, simplemente escribimos: **Recorrer Lista**), y luego de varios refinamientos del programa, especificamos cada vez más las instrucciones, hasta llegar a un punto en el que prácticamente pueden traducirse a un verdadero lenguaje de programación.

EJECUTABLES FICTICIOS

Algunos entornos de desarrollo –como el Visual Basic– dan como resultado archivos ejecutables (**.EXE**) que a simple vista parecen programas autónomos o stand-alone. Sin embargo, contienen instrucciones que debe interpretar un componente denominado run-time, que debe estar instalado en la computadora de destino para que el programa se ejecute.

ENTORNOS DE PROGRAMACIÓN Y HERRAMIENTAS

(LA ELECCIÓN DE LOS ELEMENTOS PARA PROGRAMAR)

→ Además de elegir un lenguaje a utilizar, el programador debe elegir qué entorno de trabajo y qué herramientas empleará para programar. Aquí las opciones son muy numerosas, ya que un mismo lenguaje puede ofrecer entornos muy distintos con los cuales trabajar. Las dos herramientas básicas que debe brindar un entorno de programación son el **editor de código** (en donde se escriben los programas fuente) y el **compilador** o **intérprete**, que permite ejecutar y poner a prueba los programas escritos. El entorno más rudimentario posible consta de un editor de texto común y corriente (como puede ser el Bloc de notas) y un compilador que lee los archivos de código fuente y los transforma en archivos ejecutables (**.EXE**). En el otro extremo, los entornos más completos ofrecen, por ejemplo, editores de texto "inteligentes" que corrigen la sintaxis a medida que se escribe el programa, intérpretes que permiten ejecutar los programas sin necesidad de compilarlos, organizadores de archivos o de objetos, herramientas de depuración para ejecutar los programas línea por línea, editores de pantallas o de ventanas para diseñar visualmente las interfaces de usuario, editores de menús, entre otras herramientas (**Figura 3**).

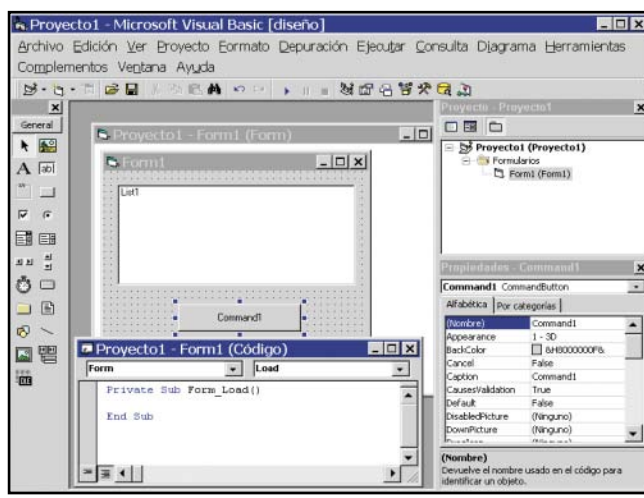


Figura 3. *El entorno de programación de Visual Basic ofrece múltiples herramientas para diseñar ventanas, depurar código, etcétera.*

EDITORES DE CÓDIGO

La elección de un buen editor de código es fundamental para el programador, ya que es la herramienta con la que trabajará la mayor parte del tiempo. Los entornos de programación más completos siempre incluyen un editor propio, el cual suele brindar algunas facilidades como indentación automática (esto significa que las líneas de código se acomodan solas, con la sangría que corresponda), verificación de sintaxis mientras se escribe, acceso rápido a instrucciones y palabras reservadas del lenguaje, etcétera (**Figura 4**).

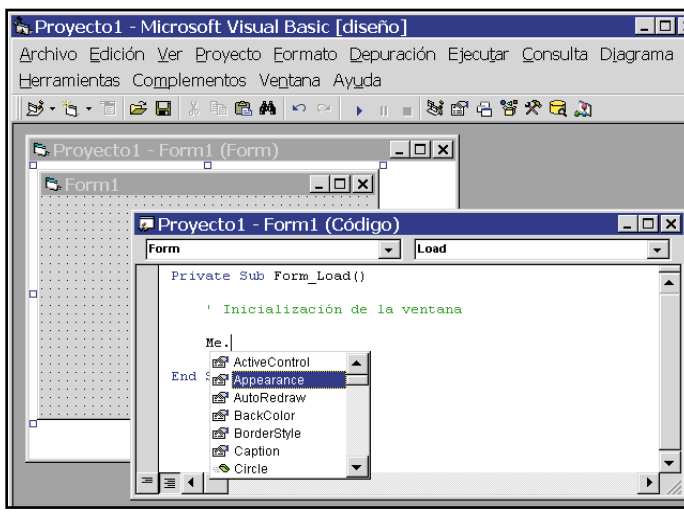


Figura 4. *Los editores de código modernos ofrecen algunas facilidades como el uso de colores para distinguir distintos elementos del código y la visualización separada de funciones y procedimientos, entre otras.*

Además de brindar un editor de código propio, los entornos de programación permiten incorporar código fuente escrito "por fuera" del entorno. De esta forma, el programador siempre tiene la posibilidad de utilizar cualquier editor de texto de su preferencia para escribir los programas, y luego incorporarlos en el entorno para probarlos y corregirlos. Los programadores más experimentados suelen utilizar editores completamente básicos como **Notepad**, un programa que viene incluido en Windows desde sus primeras versiones. Sólo para expertos.

ENTORNOS VISUALES

A partir del surgimiento del popular lenguaje Visual Basic, comenzó un concepto nuevo en programación que dio en llamarse **programación visual**. Esta forma de programar emplea elementos visuales –como por ejemplo, ventanas, botones, cuadros de texto, etc.– para diseñar los programas. Los elementos visuales, también llamados **controles**, poseen cierta "inteligencia" para saber qué hacer cuando el usuario interactúa con ellos (por ejemplo, cuando hace clic sobre un botón). De esta forma, ahorran bastante trabajo de programación y evitan la necesidad de programar al detalle cada cosa que deben hacer los controles.

A pesar de su aparente facilidad para crear programas, los entornos visuales tienen sus trampas: por utilizar elementos estándares de la interfaz gráfica del sistema operativo, deben preverse gran cantidad de eventos posibles que pueden afectar a nuestros programas. Por ejemplo, el programa debe estar preparado para que el usuario lo cierre en cualquier momento, sin importar lo que esté haciendo. También debe preverse que se pueda minimizar o maximizar la ventana del programa, o que ésta quede oculta detrás de la ventana de otro programa, y que ninguno de estos sucesos genere errores.

LIBRERÍAS

Las librerías (*library*, en inglés, que en realidad debería traducirse como "biblioteca") son repositorios de elementos listos para usar que podemos aprovechar en nuestros programas para no tener que escribir código que ya ha sido escrito por otros (o incluso por nosotros mismos). Todos los lenguajes y entornos de programación brindan alguna forma de acceso a diversas librerías. Éstas suelen agrupar elementos con similar funcionalidad; por ejemplo, es común encontrar librerías para manejo de funciones matemáticas. Si nos enfrentamos con la necesidad de escribir un programa que en algún momento calcule la raíz cuadrada de un número, entonces hacemos referencia a la librería de funciones matemáticas, y luego podremos utilizar la función para calcular raíces cuadradas, sin preocuparnos por cómo esté programada dicha función.

INTERFACES GRÁFICAS

La programación visual es ideal para crear programas que emplean la interfaz gráfica estándar del sistema operativo, ya que los controles utilizados para darles funcionalidad a los programas son elementos propios de dicha interfaz.

COMPILADORES E INTÉRPRETES
(DIFERENCIAS Y USOS)

➔ Tal como vimos anteriormente en este capítulo, el código fuente de los programas que escribimos debe traducirse en algún momento a código ejecutable, ya que de otra forma la computadora no lo podría entender y, por ende, el programa no podría ejecutarse.

Hay dos formas de traducir código fuente en código ejecutable: **compilación** e **interpretación**. Estas formas difieren en cuanto al momento en que se realiza la mencionada traducción y son los **compiladores** e **intérpretes** –programas que, por lo general, se encuentran integrados al entorno de programación– quienes las llevan a cabo. Luego veremos la diferencia entre unos y otros.

LENGUAJES DE ALTO NIVEL Y LENGUAJE DE MÁQUINA

Los lenguajes de programación que usamos habitualmente (Basic, Pascal, C, etc.) utilizan instrucciones llamadas de **alto nivel**. Al pasar un programa escrito en alguno de estos lenguajes por un compilador o un intérprete, las instrucciones de alto nivel son traducidas a instrucciones en **lenguaje de máquina**, también se llamadas instrucciones de **bajo nivel**.

En general, cada instrucción de alto nivel se traduce en numerosas instrucciones de bajo nivel. Por ejemplo, una simple instrucción de alto nivel podría ser la siguiente:

```
c := a + b;
```

(Como veremos más adelante, esto significa: sumar **a** y **b** y almacenar el resultado en **c**.)

Al traducirse a lenguaje de máquina, la instrucción anterior se convierte en una secuencia de instrucciones parecida a ésta:

```
1. buscar posición de memoria de a
2. copiar el valor en la posición de memoria de a al registro 1
3. buscar posición de memoria de b
4. sumar al valor del registro 1 el valor en la posición de memoria de b
5. buscar posición de memoria de c
6. almacenar en la posición de memoria de c el valor del registro 1
```

Si bien el ejemplo anterior no tiene el aspecto de un programa en lenguaje de máquina –dado que tales programas se componen enteramente de números **hexadecimales** y pequeñas instrucciones llamados **mnemónicos** (**Figura 5**)–, sirve para mostrar qué hacen sus instrucciones, ya que los programas en lenguaje de máquina se basan enteramente en el manejo de posiciones de memoria, registros del procesador y operaciones matemáticas elementales.

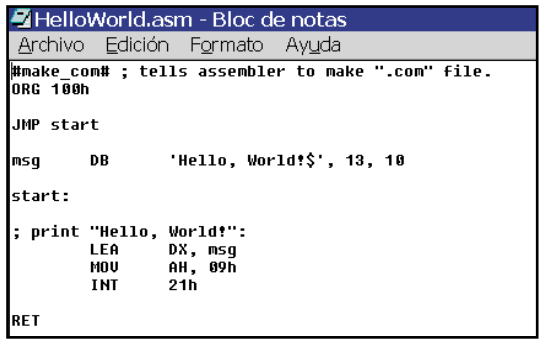


Figura 5. *Los programas en assembler (código de máquina) son muy eficientes en su ejecución, pero difíciles de entender. Aquí, la versión del programa "Hello World" en assembler.*

Si bien es posible programar directamente en lenguaje de máquina, la gran dificultad que implica esta tarea hace que siempre sea necesario recurrir a compiladores e intérpretes que permitan programar en algún lenguaje de alto nivel.

CÓDIGO FUENTE, CÓDIGO OBJETO Y CÓDIGO EJECUTABLE

Cuando se escribe un programa en algún lenguaje de alto nivel, lo que se obtiene es un programa en **código fuente**, el cual debe traducirse a lenguaje de máquina para que pueda ejecutarse. Cuando se procesa un programa en código fuente con un compilador, lo que se obtiene es un programa en **código objeto**. Este programa es "casi" un programa ejecutable, sólo falta vincularlo con las librerías externas al programa para convertirlo en **código ejecutable**.

Este proceso lo lleva a cabo una herramienta llamada **vinculador** (**Figura 6**). En los entornos de programación modernos, prácticamente no se hace distinción entre el compilador y el vinculador, dado que ambos son un proceso indivisible.

Por eso es que del código fuente se pasa directamente al ejecutable.

NÚMEROS HEXADECIMALES

El sistema hexadecimal utiliza una base de 16 dígitos para representar los números, en lugar de los 10 dígitos del sistema decimal. Para agregar los seis dígitos adicionales se emplean las letras de la **A (1)** a la **F (16)**.

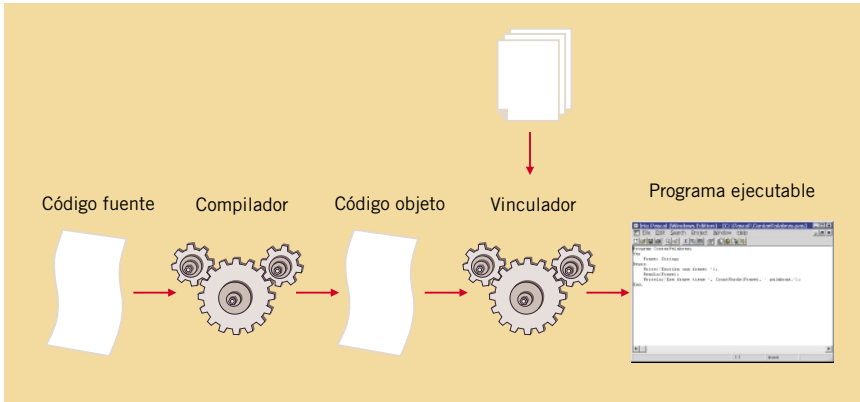


Figura 6. *El código fuente atraviesa distintas etapas hasta llegar a ser un programa ejecutable.*

DIFERENCIAS ENTRE COMPILAR E INTERPRETAR

Ya vimos que al compilar un programa obtenemos como resultado un ejecutable (en general, un archivo **.EXE**). Para probar cómo funciona, simplemente debemos ejecutarlo como a cualquier otro programa. Si al probarlo encontramos que cometimos algún error (que suele ser lo más común), tendremos que editar el código fuente, luego buscar el error, corregirlo, compilar nuevamente y volver a probar. Deberemos repetir este proceso tantas veces como sea necesario para obtener un programa tan libre de errores como sea posible. Todo el ciclo de corrección del programa, la compilación y la prueba constituye la **depuración** o *debugging* de un programa.

Para agilizar este ciclo se utilizan los **intérpretes** en lugar de los compiladores. A diferencia del compilador, el intérprete no compila todo el programa de una vez, sino que lee y compila una por una las instrucciones de un programa fuente, en el orden de ejecución. Luego de compilar cada instrucción, la ejecuta (**Figura 7**).

La utilización de un intérprete acelera el proceso de depuración por el hecho de que, para probar un programa, no necesitamos realizar la compilación del programa completo. Por lo general, el intérprete está integrado al entorno de programación y esto, en realidad, nos permite ejecutar (en forma **interpretada**) el programa que estamos escribiendo sin necesidad de dejar el entorno de trabajo. A su vez, otra ventaja que otorga el intérprete es la posibilidad de ejecutar

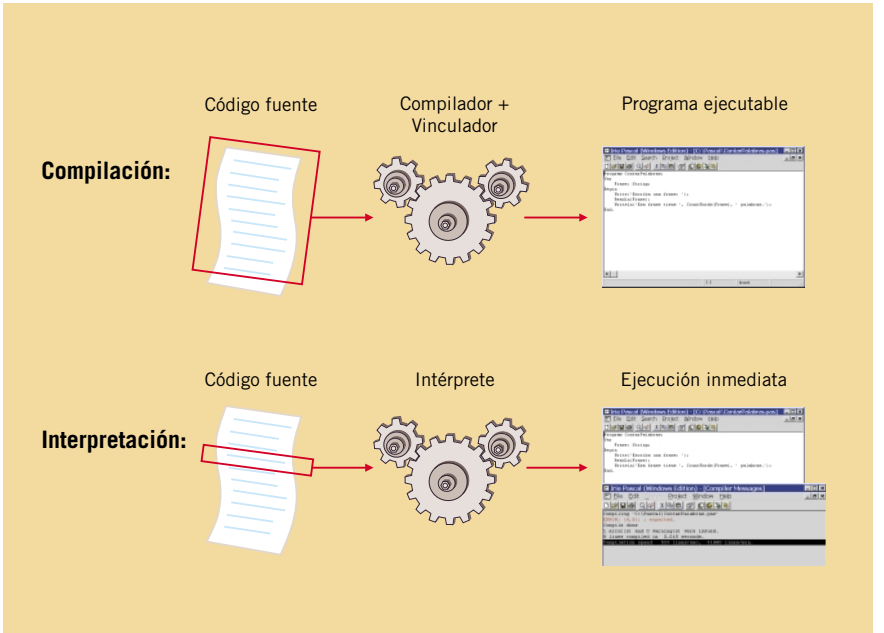


Figura 7. *Otra de las diferencias entre compiladores e intérpretes es que este último no genera como resultado ningún programa ejecutable.*

paso a paso: esto significa que, luego de compilar y ejecutar cada instrucción, el intérprete queda a la espera de que le digamos que ejecute la próxima instrucción. De esta forma, tenemos la posibilidad de verificar el estado de un programa luego de ejecutar cada instrucción, con lo cual podremos detectar fácilmente los errores. En cualquier momento podemos detener la ejecución para volver al editor de código y hacer las correcciones que sean necesarias.

Lógicamente, la ejecución de un programa en forma interpretada es más lenta que la ejecución en forma compilada, puesto que el programa compilado ya fue traducido en su totalidad a lenguaje de máquina y no es necesario traducir cada instrucción antes de ejecutarla. Es por esta razón que no se pueden hacer pruebas de la velocidad (o de la *performance*) de un programa ejecutándolo a través de un intérprete, puesto que la velocidad observada no será la velocidad real a la que puede ejecutar el programa.

PRIMEROS PASOS EN PASCAL

(DESCRIPCIÓN DEL LENGUAJE Y LA PRIMERA PRÁCTICA)

→ El lenguaje Pascal fue creado en el año 1968 como una herramienta didáctica para el aprendizaje de técnicas de programación. A pesar de sus años, Pascal sigue vigente hoy en día a través de implementaciones modernas, como el lenguaje Delphi, que es capaz de utilizarse en el entorno .NET de Microsoft.

Pascal se caracteriza por su poder para crear programas fácilmente entendibles, incluso para quien no conoce el lenguaje a fondo. Su similitud con el pseudocódigo hace que resulte fácil transformar un programa esbozado mediante este pseudolenguaje en una implementación plenamente funcional.

Por todas estas cualidades es que elegimos el lenguaje Pascal para ejemplificar los conceptos de programación vertidos a lo largo de este libro. Para crear y probar los programas, utilizaremos **Irie Pascal**, una herramienta shareware de fácil utilización que presenta una interfaz gráfica estándar de Windows, a diferencia de la mayoría de los entornos de programación en Pascal, que sólo ofrecen una interfaz de usuario basada en caracteres.

El instalador de Irie Pascal puede obtenerse en Internet, en la dirección **www.irietools.com/iriepascal/download.html**. Luego de bajar a la computadora el archivo del instalador (denominado **ipw-eval.exe**), la herramienta se instala simplemente al ejecutar dicho archivo y seguir los pasos del asistente. Una vez concluida la instalación, podrá accederse a Irie Pascal a través del menú **Inicio** de Windows, en la opción **Programas/Irie Pascal Evaluation Copy**.

OTROS ENTORNOS DE DESARROLLO EN PASCAL

Además de Irie Pascal, existen muchos entornos de desarrollo que pueden utilizarse para programar en este lenguaje, algunos más poderosos e incluso gratuitos. Uno de ellos es Turbo Pascal, la herramienta que usan muchos estudiantes universitarios, que la empresa Borland ha decidido ofrecer gratuitamente. Otro entorno que ha ganado amplia popularidad es Free Pascal, una herramienta de distribución libre y de código abierto (con licencia GNU), similar en muchos aspectos a Turbo Pascal, pero con unas cuantas mejoras. Si bien estas herramientas son más poderosas que Irie Pascal, tienen la desventaja de ser más difíciles de usar.

EL ENTORNO DE TRABAJO

Irie Pascal ofrece un entorno amigable de programación para trabajar en lenguaje Pascal. Brinda una interfaz de usuario basada en Windows, si bien los programas que genera se ejecutan en una ventana del intérprete de comandos (o una ventana de MS-DOS, según la versión de Windows que se utilice). Esto significa que los programas creados en Irie Pascal no utilizan los elementos de la interfaz gráfica de Windows, tales como ventanas, botones, barras de desplazamiento, etc. Pero esto no impide utilizarlo para aprovechar a pleno las características del lenguaje Pascal.

La pantalla principal de trabajo de Irie Pascal la constituye el editor de código (**Figura 8**). Éste no se diferencia mucho del Bloc de notas de Windows, excepto por algunas facilidades para la programación tales como indentación (sangría) automática o selección de bloques. Además, es capaz de manejar múltiples ventanas, lo cual permite trabajar sobre varios programas a la vez.

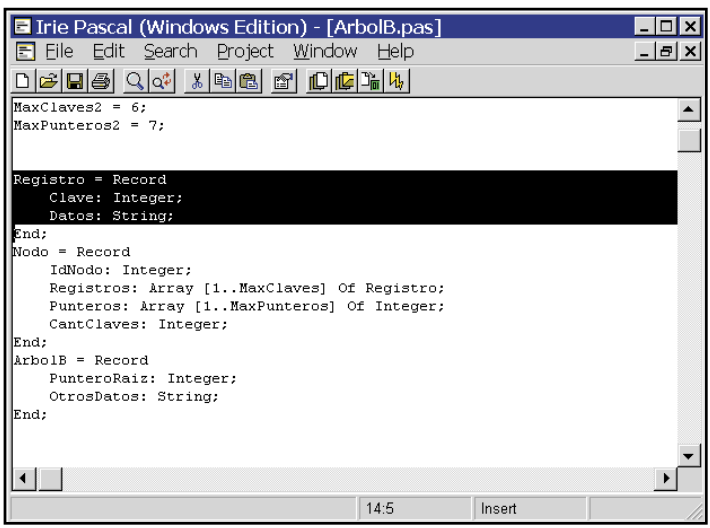


Figura 8. *El editor de código es la principal ventana de trabajo del entorno Irie Pascal.*

NUESTRO PRIMER PROGRAMA PASCAL

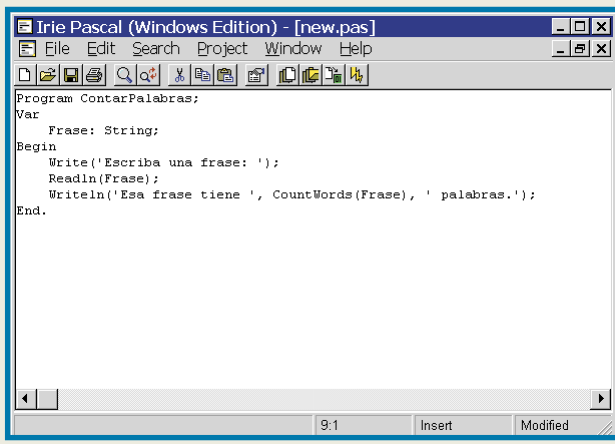
A continuación escribiremos un programa Pascal sencillo que pide al usuario que escriba una frase y luego cuenta la cantidad de palabras contenidas en ella.

PXP 01 | LA PRIMERA PRUEBA

[1] Entre en Irie Pascal y cree un archivo en blanco (**File/New**).

[2] Escriba el siguiente programa:

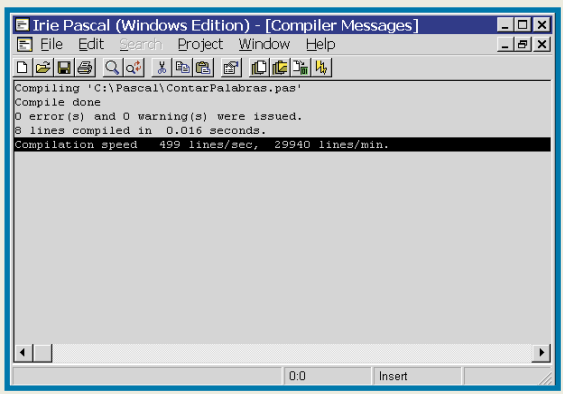
```
Program ContarPalabras;
Var
   Frase: String;
Begin
   Write('Escriba una frase: ');
   Readln(Frase);
   Writeln('Esa frase tiene ', CountWords(Frase), ' palabras.');
End.
```



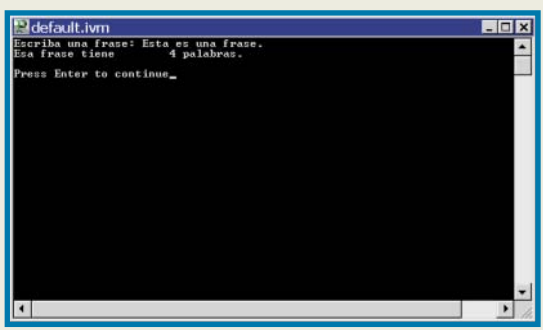
[3] Grábelo (**File/Save**) con el nombre **ContarPalabras.pas**. Tome nota de la carpeta en donde lo guarda, pues después deberá buscarlo para poder ponerlo a prueba.

[4] Acceda a la opción de menú **Project/Choose Program...**, y luego busque y seleccione el archivo **ContarPalabras.pas** en donde lo grabó en el paso 3.

[5] Acceda a la opción de menú **Project/Compile** (o directamente presione **F9**). Aparecerá una ventana que muestra los resultados de la compilación. Si en esta ventana se indica que hay errores, vuelva a la ventana de edición de código y revise el programa. Tenga en cuenta que debe ser idéntico al código escrito en el paso 2, sin omitir ni agregar ningún espacio ni signo de puntuación.



[6] Ejecute el programa accediendo a la opción de menú **Project/ Run** (o **CONTROL+F9**). Aparecerá una ventana del intérprete de comandos en la que deberá escribir una frase cualquiera y presionar **ENTER**. Luego el programa le indicará cuántas palabras tiene la frase escrita.



ACTIVIDADES DE AUTOEVALUACIÓN

Una serie de consignas y actividades para repasar los principales conceptos tratados en este capítulo y, de esta manera, afianzar el aprendizaje:

→ **1.** Trabajamos sobre un lenguaje de programación denominado "Damas", que tiene un conjunto de instrucciones muy reducido que sirve para mover una ficha sobre un tablero cuadriculado, de a una casilla por vez:

INSTRUCCIONES DEL LENGUAJE "DAMAS"

MoverArriba: mueve la ficha a la casilla superior respecto de su posición actual.
MoverDerecha: mueve la ficha a la casilla derecha respecto de su posición actual.
MoverAbajo: mueve la ficha a la casilla inferior respecto de su posición actual.
MoverIzquierda: mueve la ficha a la casilla izquierda respecto de su posición actual.

Supongamos que la ejecución de un programa escrito en el lenguaje "Damas" siempre comienza con la ficha colocada en la casilla superior izquierda del tablero. Escriba un programa en lenguaje "Damas" que desplace la ficha hasta la casilla inferior derecha, en un tablero de cuatro por cuatro.

→ **2.** Ahora contamos con una versión más potente del lenguaje, llamada "Damas++". Esta versión agrega una nueva instrucción:

INSTRUCCIÓN AGREGADA EN "DAMAS++"

RepetirHastaUltimaCasilla: repite todas las instrucciones anteriores del programa, hasta que la ficha se encuentra en la casilla inferior derecha.

Escriba un programa en lenguaje "Damas++" que desplace la ficha hasta la casilla inferior derecha en un tablero cuadrado de cualquier dimensión.

→ **3.** Escriba un programa en "Damas++" que haga que la ficha se mueva siempre en forma circular entre los cuatro casilleros superiores izquierdos del tablero.

→ **4.** ¿Cuál es la diferencia entre un compilador y un intérprete?

→ **5.** ¿Qué ventajas tienen los lenguajes de programación estructurados?