Introducción a la programación

Mihaela Juganaru Mathieu

PRIMERA EDICIÓN EBOOK

México, 2014

Para establecer comunicación con nosotros puede hacerlo por:

correo:
Renacimiento 180, Col. San Juan Tlihuaca, Azcapotzalco, 02400, México, D.F.

fax pedidos:
(01 55) 5354 9109 • 5354 9102

e-mail:
info@patriacultural.com.mx

home page:
www.patriacultural.com.mx

Dirección editorial: Javier Enrique Callejas
Coordinadora editorial: Estela Delfín Ramírez
Supervisor de prepensa: Gerardo Briones González
Diseño de portada: Juan Bernardo Rosado Solís
Fotografías: © Thinkstockphoto

Revisión técnica: Fabiola Ocampo Botello
José Sánchez Juárez
Roberto de Luna Caballero
Escuela superior de Cómputo
Instituto Politécnico Nacional
Roland Jégou
Ecole Nationale Supérieure des Mines de St. Etienne

Introducción a la programación

Renacimiento 180, Colonia San Juan Tlihuaca,
Delegación Azcapotzalco, Código Postal 02400, México, D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro núm. 43

ISBN ebook: 978-607-438-920-3

Impreso en México
Printed in México

Primera edición ebook: 2014

Contenido

Capítulo 3
Variables, apuntadores, funciones y recursividad

Capítulo 4
Arreglos, cadenas de caracteres, archivos

Capítulo 5

Estructuras de datos. Tipos abstractos

Capítulo 6

Búsqueda. Selección. Ordenamiento

Documentos adicionales que se ecuentran en el CD-ROM

Apéndice 1
Representación interna de la información

Apéndice 2
Uso del software Raptor para la elaboración de diagramas de flujo

Apéndice 3
Del código hasta el ejecutable: Compilación y uso de librerías externas

Apéndice 4
Complejidad de algoritmos: una corta introducción

Agradecimientos

A mis hijos Marceline, Virgile y Guilhem; mi luz, mi energía y mi razón de vivir. Les pido perdón por el tiempo que no estuve con ustedes para ir a caminar o para pasear en bicicleta; pero, el resultado es este libro.

Este libro es hoy una realidad y fue posible porque la ingeniera Estela Delfín Ramírez me dio la oportunidad de hacerlo y me brindó toda su confianza. Le agradezco mucho eso y que haya tenido mucha paciencia durante los meses que tardó el libro en estar listo.

Gracias a mi mamá, quien me da el coraje necesario en cada llamada que me hace desde Rumania.

También agradezco mucho a tres valiosos compañeros de trabajo: a los profesores Jean-Jaques Girardot y Roland Jégou, por su apoyo y ayuda en la consulta de la bibliografía y en el intercambio de información acerca de los lenguajes de programación y la complejidad de los algoritmos, a Jean-François Tchebanoff, por su apoyo técnico en varias ocasiones durante la preparación de este libro.

Agradezco de una forma muy especial a los profesores Fabiola Ocampo Botello, Roberto de Luna y José Sánchez Juárez de la ESCOM-IPN y a la profesora Irma Ardón de la UAM-Azcapotzalco, porque sus sugerencias, comentarios y observaciones fueron de gran ayuda para la realización de este proyecto. Una mención especial al profesor Nicolás Domínguez Vergara, quien durante su gestión como jefe del Departamento de Sistemas, en la Universidad Autónoma Metropolitana, unidad Azcapotzalco, me ofreció la inmensa oportunidad de trabajar en México.

Agradezco también a la institución que me formó en Francia: Ecole Nationale Supérieure des Mines de St. Etienne.

Un pensamiento para todos mis alumnos, tanto a los que han estado conmigo en las aulas de Francia y como en las aulas de México, durante todos los años de mi labor docente, quienes ayudaron con sus dudas, detalles técnicos, generalidades y detalles teóricos. Preguntar es un buen camino para demostrar, no que el alumno no sabe, sino para que el profesor sepa conducirse y pueda lograr una mejor transmisión de sus conocimientos.

Gracias a mi esposo, mi hermana y a algunos de mis amigos, quienes soportaron y toleraron mis ausencias y mi falta de disponibilidad.

Presentación

¡Nadie nace sabiendo programar computadoras!

La programación es un conocimiento que se aprende, como se aprende hacer reacciones químicas en un laboratorio, resolver ecuaciones matemáticas o andar en bicicleta. El principal objetivo de este libro es mostrar que el aprendizaje de la programación puede ser fácil, si se empieza desde lo básico y se continúa de manera gradual, hasta que se es capaz de escribir un programa que resuelve un problema.

La primera dificultad del aprendizaje de la programación radica en la necesidad de aprender dos cosas bastante diferentes de manera simultánea:

1. Un lenguaje para transmitir a la máquina las órdenes que se le quieren dar; esto es, el lenguaje de programación y una manera de pensar y concebir dar órdenes a la computadora.
2. El algoritmo traducido en programa.

El uso de un lenguaje siempre debe respetar un conjunto de reglas de sintaxis y de semántica; sin embargo, un programa que es correcto desde el punto de vista del lenguaje no siempre va a realizar la tarea o a resolver el problema que se quiere solucionar.

En el proceso de aprendizaje de un programa, si ya se conoce programar con algún otro lenguaje de programación, resulta más fácil aprender otro lenguaje, sin muchas explicaciones, mejor aún si el segundo lenguaje pertenece al mismo paradigma de programación. Por esta razón, considero que el lenguaje C es el más óptimo de aprender como primer lenguaje, ya que es un lenguaje imperativo, que permite un manejo muy preciso de conceptos importantes de la programación, como las variables, los apuntadores, las funciones y los arreglos, entre otros aspectos. Cualquier otro lenguaje de programación, como Pascal, los lenguajes de los software MATLAB o SCILAB o el lenguaje R, los lenguajes C++ o Java, deberán aprenderse después de que se conoce y se maneja de manera óptima el lenguaje C que, como ya se dijo antes, es mucho más fácil de entender y manejar.

Este libro consta de seis capítulos que lo guiarán en su aprendizaje en el conocimiento de la programación; desde los aspectos básicos, las bases de la programación hasta llegar a los conceptos más difíciles de la programación, como los arreglos, las cadenas de caracteres y los archivos, las estructuras avanzadas de datos, y lo más importante de la programación: el ordenamiento, la búsqueda y la selección de los lenguajes más complejos. Toda esto se describe y analiza en cada uno de los seis capítulos de los que consta el libro:

1. Del algoritmo al programa
2. Programación en Lenguaje C: Conceptos
3. Variables, apuntadores, funciones y recursividad
4. Arreglos, cadenas de caracteres, archivos
5. Estructuras de datos. Tipos abstractos
6. Búsqueda, Selección Ordenamiento

Para complementar el aprendizaje de la programación, el libro está acompañado de un CD-ROM de apoyo, el cual contiene mucha información adicional, que le será de gran utilidad, como: la descripción de todos los programas que se estudian en el libro y funciones o fragmentos de códigos. Cabe resaltar que en el CD-ROM se aborda cada uno de los capítulos del libro de manera independiente. Asimismo, también contiene información adicional, como algunas animaciones escritas en Java por los algoritmos de ordenamiento. También incluye cuatro apéndices en los que se abordan y resuelven problemas conexos tratados en el libro, como: la representación de la información, el uso de un software para realizar diagramas de flujo, detalles prácticos del uso softwares libres para programar en lenguaje C y un último apéndice sobre la complejidad de los algoritmos.

La realización de este libro es la síntesis de mi trabajo como profesora de computación desde el año 2000 hasta el 2008 en Francia y de los tres trimestres que cursé en la Universidad Autónoma Metropolitana, unidad Azcapotzalco.

Quiero agradecer una vez más a todo el equipo de Grupo Editorial Patria por su confianza y su apoyo continuo, así como por su eficiencia.

Este libro está dirigido, en primer lugar, a aquellos alumnos de las escuelas de carreras técnicas y científicas, quienes tienen que cursar obligatoriamente la asignatura de introducción a la programación, y en segundo lugar a los alumnos universitarios que cursan una carrera de computación, para ellos, la lectura a profundidad de los últimos tres capítulos les será de gran utilidad.

El libro puede ser utilizado con toda confianza por los profesores de computación como un soporte metodológico o como una colección de problemas resueltos, con el fin de utilizarlos en clase o como una amplia colección de problemas propuestos para tareas o exámenes de evaluación.

La autora

Prólogo

El libro *Introducción a la Programación* ofrece las generalidades básicas que deben conocer aquellas personas que deseen introducirse en la programación de computadoras; pues, mediante aspectos visuales, como la presentación de diagramas de flujo, y de aspectos secuenciales, como los algoritmos, la autora expone al lector la lógica de la programación estructurada, para luego, de manera ordenada, explicar, las generalidades básicas del lenguaje de programación C, a través de ejemplos concretos, así como los temas que resultan esenciales, básicos y necesarios para el desarrollo de programas de computadoras en este lenguaje de programación. Asimismo, introduce al lector en el estudio de temas de programación de computadoras más avanzados, como las estructuras de datos, las formas de ordenamiento, la selección y la búsqueda.

La motivación que da surgimiento a este libro, deriva de la experiencia y el deseo de la autora de ofrecer un material educativo que incorpore desde los aspectos algorítmicos hasta la puesta en marcha de un programa de computadora escrito en lenguaje C, a través de ejemplos expresados de forma didáctica, lo cuales permiten a estudiantes y profesores practicar la ejecución de los diagramas de flujo desarrollados, así como los resultados de los programas de computadora escritos en lenguaje C, mediante una explicación clara y sucinta de los ejemplos resueltos a lo largo del libro.

Una de las principales asignaturas en las que puede ser utilizado este libro es: Introducción a la programación de computadoras, debido a que presenta, de manera general, la lógica de desarrollo de diagramas de flujo y algoritmos, proporcionando la referencia de una herramienta computacional, que permite a los estudiantes el análisis de la lógica de solución de problemas, antes de la codificación. Otra de las asignaturas en la que resulta de gran utilidad este material es: Programación en lenguaje C, ya que aborda desde las estructuras básicas hasta los temas esenciales que son parte fundamental de este lenguaje, como los apuntadores y el manejo de flujos de entrada y salida.

La introducción a la comprensión de las estructuras de datos, métodos de selección, ordenamiento y búsqueda, son aspectos necesarios para el estudio de las estructuras de datos. Asimismo, incorpora temas y ejemplos relacionados con los sistemas numéricos, necesarios para la comprensión de las formas de almacenamiento y el manejo de la memoria dinámica de la computadora.

El contenido de este libro se estructura en seis capítulos:

1. Del algoritmo al programa
2. Programación en lenguaje C: Conceptos básicos
3. Variables, apuntadores, funciones y recursividad
4. Arreglos, cadenas de caracteres, archivos
5. Estructuras de datos. Tipos abstractos
6. Búsqueda. Selección. Ordenamiento

En cada uno de los seis capítulos que conforman este libro, la autora expone los aspectos teóricos y prácticos, mediante ejemplos resueltos y completos, además de que también presenta una guía de estudio para el estudiante para el desarrollo de programas, que le facilita el aprendizaje de la programación de computadoras y reforzar los aspectos teóricos y prácticos expuestos en esta obra.

Fabiola Ocampo Botello

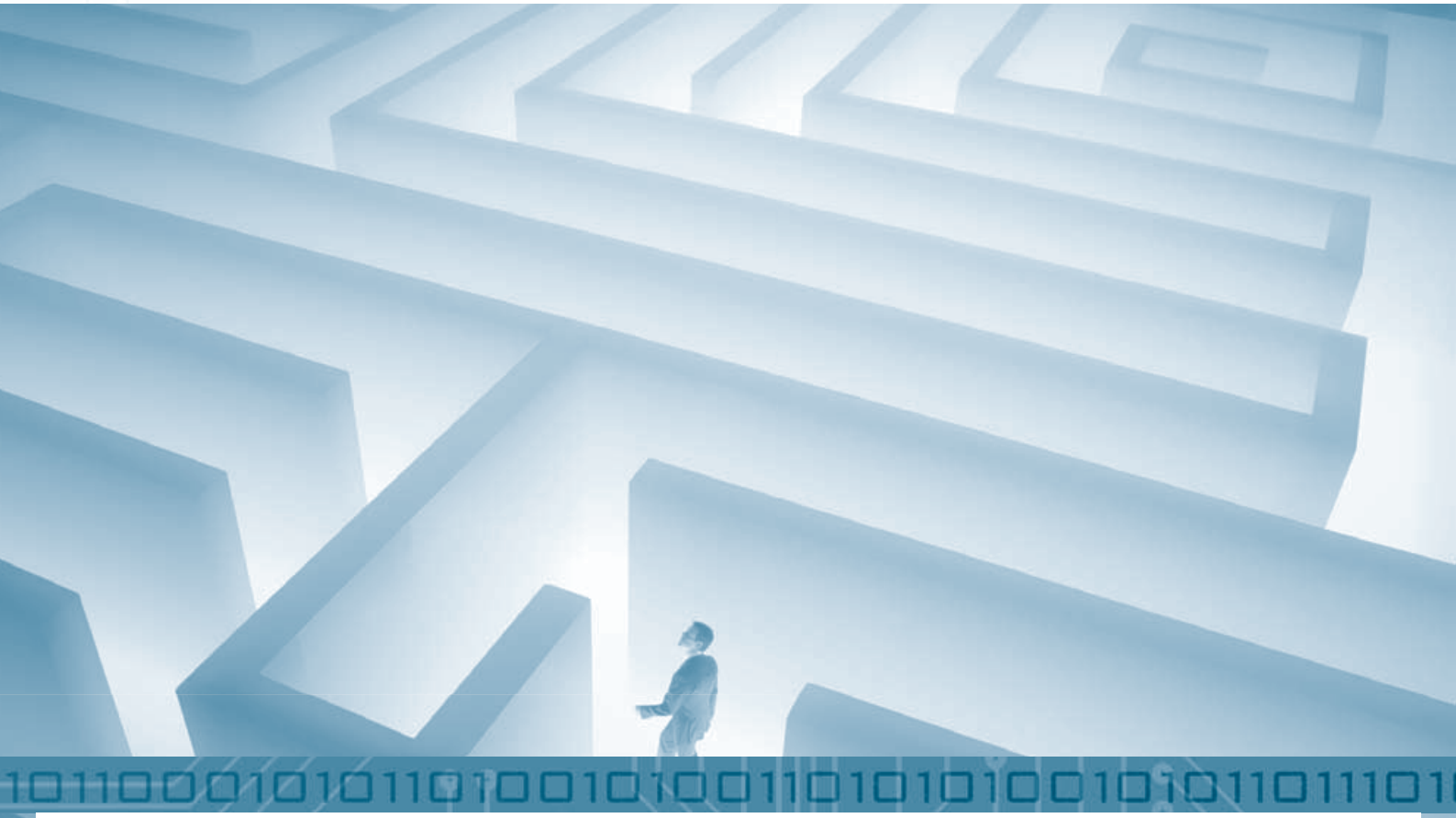
Departamento de Ingeniería en Sistemas Computacionales

Escuela Superior de Cómputo

Instituto Politécnico Nacional

México, D.F.

1 Del algoritmo al programa



Contenido

Objetivos

- Comprender que la computadora funciona con programas y que además es capaz de realizar programas.
- Conocer y comprender las nociones fundamentales de la programación: variable, tipo y función.
- Estudiar los principales paradigmas de la programación.
- Proponer algoritmos simples para la resolución de diversas tareas.
- Expresar algoritmos con diagramas de flujo o con pseudocódigo.

1.1 Programa, algoritmo, lenguaje

De acuerdo con la naturaleza del funcionamiento de las computadoras, se dice que estas siempre ejecutan órdenes en un formato que les resulta inteligible; dichas órdenes se agrupan en **programas**, conocidos como **software**, el cual, para su estudio, a su vez, se divide en dos partes: el formato de representación interno de los programas, que constituye el **lenguaje máquina** o **código ejecutable,** y el formato de presentación externa, que es un archivo o un conjunto de archivos, que puede o no estar en un formato que puede ser visto/leído por el usuario (es decir, en un formato que respeta las reglas).

Para ejecutar lo que el usuario desea hacer en su computadora, o bien para resolver un problema específico, este precisa buscar un software que realice o ejecute con exactitud la tarea que se ha planteado o elaborar y desarrollar (escribir) un programa que la realice. El trabajo de elaboración de un programa se denomina "programación". Pero la programación no es solo el trabajo de escritura del código, sino todo un conjunto de tareas que se deben cumplir, a fin de que el código que se escribió resulte correcto y robusto, y cumpla con el objetivo o los objetivos para los que fue creado.

Las afirmaciones que se derivan de lo anterior son varias:

- Conocer las herramientas, los formalismos y los métodos para transformar un problema en un programa escrito en un lenguaje (que posiblemente no será el lenguaje máquina), y para que dicho programa pueda ser transformado en un código ejecutable.
- Saber transformar el problema inicial en un **algoritmo** y luego en un programa.

La primera afirmación es genérica y se considera para varias categorías de problemas para resolver. Por su parte, la segunda es específica de un problema determinado que se tiene que resolver, para lo cual existen diversas metodologías específicas de resolución para este tipo de problemas. Para los casos de problemas muy generales, en ocasiones existen métodos conocidos que solo se adaptan a un problema en particular. El método es, por lo general, un algoritmo o una técnica de programación.

Algoritmo

Un **algoritmo** constituye una lista bien definida, ordenada y finita de operaciones, que permite encontrar la solución a un **problema determinado.** Dado un *estado inicial* y *una entrada*, es a través de *pasos sucesivos* y bien definidos que se llega a un *estado final*, en el que se obtiene una solución (si hay varias) o la solución (si es única).

Ejemplo

Problema: Gestionar la lista de compras que una empresa realiza durante un mes.

Solución

Para resolver este problema de gestión muy general, se cuenta con las herramientas que se utilizan en otros problemas que ya tienen una resolución en la empresa (por ejemplo, un programa en Java con los datos en una base de datos). Así, para la solución del problema planteado, se proponen dos opciones:

- Usar la base de datos de trabajo para guardar, también en esta, la lista de productos que se requiere comprar.
- Guardar una lista en entradas que se actualiza cada vez que se captura o se incluye un nuevo producto que la empresa necesita, y que se borra o elimina al momento que el producto ya está abastecido, y en salidas, cada vez que algún empleado necesite una impresión de dicha lista.

En este ejemplo, el algoritmo global de resolución se compone de diversos pasos sucesivos de diálogo con el usuario (un empleado de la empresa), para mantener actualizada la lista de productos necesarios; asimismo, en pasos siguientes se precisa hacer una inserción y/o una eliminación o borrado de los productos (elementos) de la lista o una impresión en una forma legible.

Un algoritmo puede ser expresado en:

- Lenguaje natural (a veces, este no resulta muy claro, pero es muy útil para problemas simples)
- Pseudocódigo
- Diagramas de flujo
- Programas

El uso de algún elemento de la lista anterior para la expresión de un algoritmo, se hace según el nivel de descripción de dicho algoritmo. Es evidente que el **lenguaje natural** es de mayor utilidad para transmitir las ideas del algoritmo. Al contrario, un **programa** es difícil de entender por simple lectura, aun por una persona que conoce el lenguaje del programa, e imposible para aquellas que no lo conocen.

El **pseudocódigo** y los **diagramas de flujo,** en cambio, se sitúan en un punto intermedio de comprensión, entre el lenguaje natural y un programa. Estas dos herramientas poseen un poder de expresión equivalente; no obstante, los diagramas de flujo tienen la ventaja de ser más gráficos y visuales.

Con base en el ejemplo anterior, se puede afirmar que la parte de solución expresada en lenguaje natural tiene algunas ambigüedades para el usuario que no es el programador; por ejemplo, ¿qué significa la expresión "de pasos sucesivos de diálogo con el usuario"? Aunque, en ocasiones, también presenta ambigüedades hasta para el propio programador; por ejemplo, ¿cuáles son "los datos en una base de datos"?, ¿una base de datos es relacional o de otro modelo?, ¿cuál interfaz?, ¿cómo se manejan las lecturas/escritura en dicha base de datos?

Las respuestas a las interrogantes anteriores se expresan de la siguiente forma:

- La primera ambigüedad ("pasos sucesivos") se debe expresar lo más detallada posible por el destinatario del programa (el usuario).
- Los otros cuestionamientos son de detalles técnicos.

La descripción de un algoritmo usualmente se realiza en tres niveles:

1. **Descripción de alto nivel.** El primer paso consiste en la descripción del problema; luego, se selecciona un modelo matemático y se explica el algoritmo de manera verbal, posiblemente con ilustraciones, pero omitiendo detalles.
2. **Descripción formal.** En este nivel se usa un pseudocódigo o diagrama de flujo para describir la secuencia de pasos que conducen a la solución.
3. **Implementación.** Por último, en este nivel se muestra el algoritmo expresado en un lenguaje de programación específico, o algún objeto capaz de llevar a cabo instrucciones.

Para llegar a la implementación, primero se deben tener descripciones de alto nivel o formalmente explícitas, sobre todo cuando el trabajo de desarrollo de un algoritmo se hace en grupo.

Análisis y comprensión de un problema

En el ejemplo que se presentó acerca de la necesidad de una empresa de gestionar la lista de compras que efectúa durante un mes, se realizó, de forma informal y muy esquemática, la presentación de un problema que ocurre comúnmente, y se indicó, de forma muy literal, cómo se puede resolver, aunque sin bastantes detalles. No obstante, también es claro que para la resolución de este problema debemos saber cómo insertar, borrar u ordenar los elementos de una lista. Así, para cada aspecto del problema se debe buscar un algoritmo que lo resuelva; por ejemplo, un algoritmo de inserción, otro para borrar de la lista un elemento y, si la lista no está explícita en la memoria de la computadora, un algoritmo para ordenar los elementos en una forma deseada.

Por lo general, un problema se descompone en subproblemas; por tanto, un algoritmo expresa la resolución de un problema (elemental o no).

Las etapas de desarrollo de un algoritmo, con base en la lógica, son las siguientes:

1. **Definición.** En esta etapa se especifica el propósito del algoritmo y se ofrece una definición clara del problema por resolver. Además, aquí también se establece lo que se pretende lograr con su solución.
2. **Análisis.** En este punto se analiza el problema y sus características, y se determinan las entradas y salidas del problema. De igual modo, también se realiza una investigación sobre si ya se conoce alguna o varias soluciones de este. En el caso de que ya se conozcan varias soluciones, entonces se determina cuál es la más conveniente para el problema que estamos tratando. Si no se conoce ninguna, o no nos satisfacen las soluciones existentes, se propone una nueva.
3. **Diseño.** Aquí es donde se plasma la solución del problema. Con ese fin, se emplea una herramienta de diseño, que consiste en el diagrama de flujo y el pseudocódigo.
4. **Implementación.** En este último paso es donde se realiza o se ve concretado el programa y, por ende, se hacen varias pruebas.

En cada una de las etapas especificadas antes, se utiliza un tipo de descripción conveniente e inteligible para cada uno de los participantes en el proceso de concepción y realización del algoritmo. Hoy en día, existe una rama de las ciencias de la computación que se ocupa del manejo de los proyectos de desarrollo de programas: la ingeniería de software.[1]

En el ejemplo citado antes, en el que se plantea el desarrollo de un programa de lista de compras, el **enunciado** constituye la definición del problema, mientras que la **fase de análisis** pone en evidencia las entradas y las salidas, el modo operativo, el formato de la base de datos y su ubicación, los dispositivos de acceso a los datos contenidos y algunos datos de volumetría.

Entre los principales datos del ejemplo, que serían tratados en esta fase, destacan: el tamaño máximo posible de la lista de compras y el tamaño del catálogo de productos, en donde el usuario deberá buscar el producto que se inserta en la lista de compras. En el diagrama de la figura 1.1 se representa un diagrama de la actividad posible del usuario.

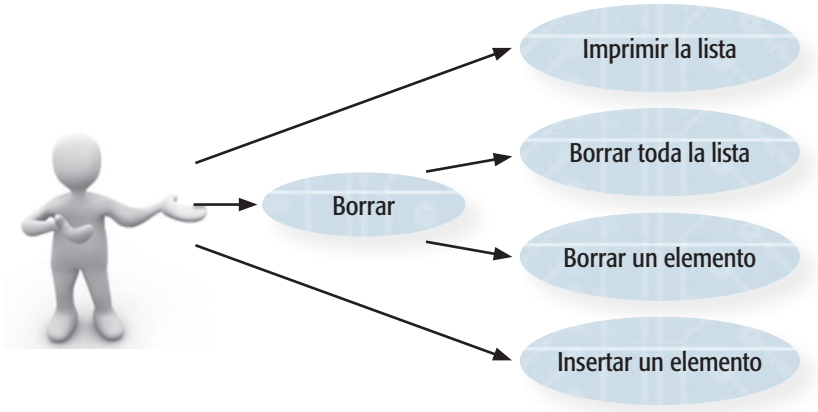


Figura 1.1

Asimismo, en esta fase es fácil distinguir la necesidad de desarrollar una manera de guardar la lista de compras de manera permanente; idealmente, esta puede generarse en un día. Así, al iniciar las labores del día se puede cargar o capturar la lista y durante el transcurso de la jornada laboral se pueden hacer diversas actualizaciones (inserción y borrado). En esta fase de análisis, también se indica el funcionamiento global del programa.

Para el mismo ejemplo, en la **fase de diseño** se planteará un diagrama de flujo de la totalidad del programa; además de que aquí también se pone en evidencia la solución que se eligió para guardar los nuevos datos (la lista de compra) y qué formato se utilizará.

Ejemplo

Supóngase un número entero N del que se requiere probar si es divisible o no entre 3.

[1] Para mayor información véase el libro Sommerville, Ian, *Ingeniería de software*, 6a. ed., Pearson Educación, México, 2001.

En este caso, la definición del problema es el enunciado mismo: "Probar si un número entero N es o no divisible entre 3". Este caso se trata de un problema muy simple de aritmética.

En la etapa de análisis, identificamos las entradas y las salidas:

Entrada: *Un número entero N.*

Salida: *Una respuesta (Sí o No).*

Para la resolución del problema de este ejemplo, conocemos la definición de la divisibilidad: "un número N es divisible entre otro número k, si la división N ÷ k es exacta (o el resto es 0)".

Asimismo, existen métodos que presentan diferentes grados de dificultad para un ser humano:

- *Realizar la división* n ÷ k *y comprobar si es exacta.*
- *Efectuar la suma de las cifras que componen el número en base 10 y verificar si el número es divisible entre 3.*

De acuerdo con la naturaleza del ser humano, él puede aplicar con mayor facilidad el segundo método, debido a que la división de la suma de las cifras y el cálculo mismo de la suma son más simples que la división inicial; sin embargo, para la computadora es lo mismo realizar la división de 78564589 entre 3 o la división de 52 entre 3. En el segundo caso, en cambio, es necesario hacer la extracción de las cifras y luego la suma de las cifras; entonces, la resolución del problema es simple, como lo establecen los siguientes pasos:

1. *Se hace la lectura del número* N.
2. *Se toma el resto de la división de* N *entre 3 (la operación módulo* N %3*).*
3. *Según el valor del resto, se escribe: "Sí" o "No".*

En la etapa de **fin de análisis**, los pasos a seguir resultan muy claros; en tanto, en el paso de **diseño** se formalizan aún más y lo describen sin ninguna ambigüedad. Durante la **implentación** (la última etapa), es preciso saber cómo introducir los valores de entrada en la computadora y cómo hacer el programa.

En el siguiente apartado se estudia cuáles son dichos valores de entrada, qué es un lenguaje de programación, qué significa programa y cómo se transforma un programa en código máquina.

Programas y paradigmas de programación y lenguajes

Un **programa informático** se define como un conjunto de instrucciones que, una vez ejecutado, realiza una o varias tareas en una computadora. De esta forma, sin programas, una computadora no puede realizar las actividades para las que fue diseñada y creada.

El conjunto general de programas que posee una computadora se denomina **software**, término que se utiliza para definir al equipamiento o soporte lógico de una computadora.

Un programa se escribe con instrucciones en un **lenguaje de programación**, el cual, a su vez, está definido por su sintaxis, que establece e indica las reglas de escritura (la gramática), y por la semántica de los tipos de datos, instrucciones, definiciones, y todos los otros elementos que constituyen un programa.

Un lenguaje de programación es un caso particular del lenguaje informático; este último permite hacer programas, pero también describir datos, configuraciones físicas y protocolos de comunicación entre equipos y programas.

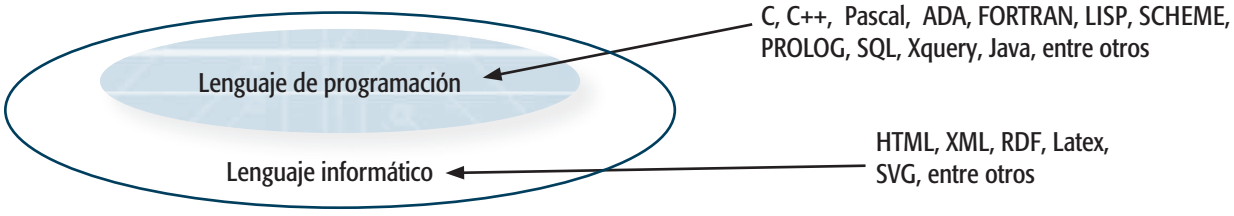


Figura 1.2 Tipos de lenguajes.

Si un programa está escrito en un lenguaje de programación comprensible para el ser humano, se le llama **código fuente**. A su vez, el código fuente se puede convertir en un archivo ejecutable (código máquina) con la ayuda de un **compilador**, aunque también puede ser ejecutado de inmediato a través de un **intérprete**.

A su vez, un **paradigma de programación** provee (y determina) la visión y los métodos de un programador en la construcción de un programa o subprograma. Existen diferentes paradigmas que derivan en múltiples y variados estilos de programación y en diferentes formas de solución de problemas:

- **Paradigma imperativo**

  En este paradigma se impone que cualquier programa es una secuencia de instrucciones o comandos que se ejecutan siguiendo un orden de arriba hacia abajo; este único enlace del programa se interrumpe exclusivamente para ejecutar otros subprogramas o funciones, después de lo cual se regresa al punto de interrupción.

- **Paradigma estructurado**

  Este paradigma es un caso particular de paradigma imperativo, por lo que se imponen únicamente algunas estructuras de código, prohibiendo una continuación del cálculo de manera caótica. Por ejemplo, se impone que las instrucciones sean agrupadas en bloques (procedimientos y funciones) que comunican; por tanto, el código que se repite tiene la forma de un ciclo (*loop*, en inglés), gobernado por una condición lógica.

- **Paradigma declarativo**

  Un programa describe el problema a solucionar y la manera de resolverlo, pero no indica el orden de las acciones u operaciones que se deben seguir. En este caso, hay dos paradigmas principales:

  - **Paradigma funcional:** Conforme a este, todo se describe como una función.
  - **Paradigma lógico:** De acuerdo con este, todo se describe como un predicado lógico.

Un problema a resolver se expresa como una llamada de una función o un predicado lógico, y su resolución depende de la descripción introducida en las funciones o los predicados.

- **Paradigma orientado a objetos**

  Existen tres principios fundamentales que gobiernan este tipo de programación:

  - **Encapsulación:** En este principio se encapsulan datos, estados, operaciones y, en ocasiones, también eventos, en objetos. El código sería ejecutado, entonces, según la ocurrencia de eventos o de creación/destrucción de instancia de objetos.
  - **Prototipos, clases y herencias:** El prototipo y la clase son las abstracciones del objeto; otros prototipos se definen de acuerdo con un prototipo existente.
  - **Tipificación y polimorfismo:** Constituyen la comprobación del tipo con respecto a la jerarquía de las clases.

- **Paradigma de programación por eventos**

  Un programa se concibe como una iteración infinita con dos objetivos: detectar los eventos y establecer el cálculo capaz de tratar el evento.

- **Paradigmas paralelo, distribuido y concurrente**

Un programa no se realiza con una sola unidad de cómputo, sino que emplea varias unidades de cálculo (reales en caso paralelo y distribuido), las cuales pueden ser procesadores o computadoras y/o unidades centrales del mismo procesador. En el caso de este paradigma, el programa se corta en subprogramas o rutinas que se ejecutan de manera independiente sobre otras unidades de cómputo, ya sea de modo síncrono o asíncrono, compartiendo o no la misma memoria.

Un lenguaje de programación puede verificar uno o más paradigmas. Por ejemplo, el lenguaje Java comprueba el paradigma orientado a objetos y el código que compone la parte de métodos de los objetos verifica el paradigma estructurado. Por su parte, el lenguaje de programación de páginas de Internet, JavaScript, funciona/trabaja conjuntamente con las páginas y el servidor del sitio; por tanto, es un lenguaje, inspirado por Java, que comprueba el paradigma de programación orientado a objetos, al tiempo que también funciona según el paradigma de la programación por eventos.

Algunos ejemplos de **lenguajes de programación imperativos** son: lenguaje máquina, lenguaje ensamblador, C, Fortran, Cobol, Pascal, Ada, C++, C#, Java. A excepción del lenguaje máquina y el lenguaje ensamblador, los otros constituyen lenguajes estructurados.

Entre los **lenguajes declarativos** más conocidos son: LISP (Scheme), Prolog, SQL, Smalltalk, Datalog. Asimismo, el lenguaje Java también puede ser considerado como un lenguaje declarativo.

Como **lenguajes orientados a objetos** existen: Simula, C++, Java, C#(.Net), Python.

Históricamente, las primeras computadoras se programaban manualmente (de forma física), cambiando los dispositivos físicos del equipo de cómputo; por ejemplo, la máquina analítica de Charles Babbage, programada por Ada Byron, o la computadora ENIAC.

Al principio, en los albores de la computación, se introdujo el lenguaje ensamblador, que codifica, con códigos literales, las operaciones del procesador, los registros y las direcciones de memoria. En la actualidad, algunas máquinas virtuales aún se pueden programar en un lenguaje ensamblador adaptado. Otro dominio actual, por el cual se utiliza el lenguaje ensamblador, es el desarrollo de interfaces específicas con dispositivos de entrada/salida de los datos. La principal ventaja del lenguaje ensamblador es un código eficaz, muy cercano al lenguaje máquina. En tanto, las principales desventajas o defectos que presenta el lenguaje ensamblador son, en principio, su "verbosidad", esto es, para escribir cálculos, que parecen simples, se escriben páginas y páginas en el lenguaje ensamblador, y la dificultad de corregir los errores que pueden parecer errores de concepción del programa o errores de compilación.

Un gran avance en materia de programación fue la aparición de los lenguajes de programación de alto nivel, por medio de los cuales se simplificó la escritura de código.

En el siguiente esquema se observa un fragmento de un programa escrito en lenguaje C, una parte del código en lenguaje ensamblador y una imagen de la memoria que contiene el código máquina.

```
int main()  LFB2:                0x100000f20 <main+4>:  0x0afc45c7  0xc7000000  0x0023f845  0x458b0000
{           pushq %rbp

int a=10;   LCFI0:               0x100000f30 <main+20>: 0xfc4503f8  0xb8f44589  0x00000001  0x25ffc3c9

int b, c;   movq  %rsp, %rbp     0x100000f40 <dyld_stub_exit+2>:    0x000000f4  0xe51d8d4c  0x41000000
                                 0xd525ff53
b = 35;     LCFI1:

c = a + b;  movl  $10, -4(%rbp)  0x100000f50 <stub helpers+12>:     0x90000000  0x00000068  0xffe6e900
                                 0x0000ffff
return 1;   movl  $35, -8(%rbp)
                                 0x100000f60:           0x00000001  0x0000001c  0x00000001  0x00000020
}           movl  -8(%rbp), %eax
                                 0x100000f70:           0x00000000  0x00000020  0x00000002  0x00000000
            addl  -4(%rbp), %eax
                                 0x100000f80:           0x00000000  0x00000038  0x00000038  0x00001001
            movl  %eax, -12(%rbp)
                                 0x100000f90:           0x00000000  0x00000038  0x00000003  0x0003000c
            movl  $1, %eax

            leave

            ret
```

En la corta historia de la computación (corta en comparación con otras ciencias y áreas del conocimiento humano), han sido propuestos varios lenguajes, pero solo algunos cuantos han sido utilizados en realidad.[2]

En la figura 1.3 se observa una lista de lenguajes de programación, ordenados cronológicamente (en azul se destacan los lenguajes de descripción de datos más importantes y el protocolo fundamental de Internet):[3]

[2] Algunos lenguajes, como ALGOL, para la programación, o SGML, para la descripción de los datos, fueron propuestos; sin embargo, técnicamente, nunca se desarrollaron el compilador ni las herramientas necesarias para trabajar con la versión completa. Estos lenguajes se consideran importantes por su incursión en la historia de la computación y porque constituyen el origen de otros lenguajes de programación (como PASCAL) o HTML y XML.

[3] Consultar la página http://oreilly.com/news/graphics/prog_lang_poster.pdf, para observar un esquema que aborda la historia de los lenguajes, sus versiones y su filiación.

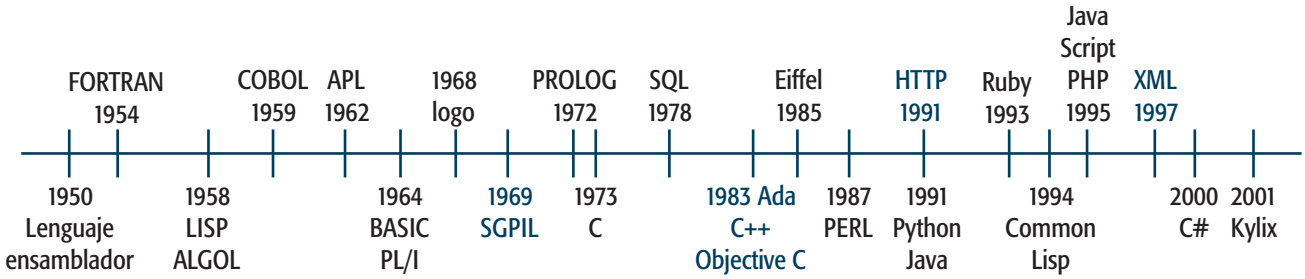


Figura 1.3 Línea de tiempo de los lenguajes de programación.

Esta proliferación y riqueza de lenguajes de programación tiene su origen en:

- El importante desarrollo de software, el cual, cada dos años, ofreció un poder de cálculo multiplicado y de almacenamiento de datos por *n*, por el mismo precio.
- La diversificación de los campos de aplicación. En un principio, la mayor necesidad de los lenguajes de programación era tratar grandes volúmenes de datos e importantes cálculos numéricos; sin embargo, las necesidades cambiaron, por lo que después aparecieron aplicaciones de inteligencia artificial, de manejo de bases de datos, de tratamiento y de generación de imágenes.
- La teoría de la computación en un amplio sentido. Por ejemplo, los dos casos siguientes:
  - La teoría de Codd, de álgebras relacionales (creada en la década de 1970), que permitió el desarrollo del lenguaje SQL para el manejo de las bases de datos relacionales.
  - El trabajo de MacCarthy (1956) sobre las funciones recursivas, que permitieron el desarrollo del lenguaje LISP.
- Las nuevas metodologías de ingeniería de software. Aquí, lo más importante es el uso extendido del paradigma orientado a objetos.
- La implementación. El uso práctico de un lenguaje permite distinguir las limitaciones de uso e impulsa las nuevas proposiciones para su mejoramiento.

Hoy en día, aún se trabaja en el desarrollo de lenguajes de programación, pero desde dos perspectivas básicas: proponer nuevas soluciones a los problemas actuales[4] y mejorar algunos de los lenguajes actuales, proponiendo nuevos estándares.

En la actualidad, el uso de un lenguaje de programación está condicionado por:

- El conocimiento del lenguaje en cuestión; es decir, su sintaxis y la semántica de los conceptos y las instrucciones que lo componen.
- El tipo de problema a resolver. Por ejemplo, para consultar datos que se guardan en un formato específico en una base de datos o en una base de conocimiento se utilizan, comúnmente, los lenguajes de tipo declarativo, donde se caracterizan los datos que se esperan en salida, como SQL para la base de datos relacional, PROLOG para la base de conocimiento, XQuery y XSLT para colecciones de datos en el formato XML. En otro ejemplo, para dar las órdenes de instalación de software, es conveniente escribir programas en el *shell* del sistema operativo.
- El derecho y la posibilidad material de utilizar un compilador o intérprete de dicho lenguaje, ya que estos tipos de software (compilador, taller de desarrollo, intérprete) suelen tener un costo monetario o licencias restrictivas.
- La configuración física que está disponible. Por ejemplo, si está disponible una arquitectura multiprocesador, sería más conveniente utilizar un lenguaje de tipo C o FORTRAN, por medio de los cuales se abstendría de realizarse el cálculo paralelo, o emplear herramientas de paralelización automática. En el caso de que el programa tuviera que

[4] Por ejemplo, un grupo de trabajo de W3C aún trabaja en el desarrollo de un lenguaje de manejo y actualización de colecciones de archivos XML.

explorar y comunicar con una interfaz de un equipo raro, como una máquina de producción o un dispositivo de medición, es preferible escribirlo en un código del lenguaje ensamblador.

- La configuración del software que está disponible o que se impone por la construcción del programa y el uso ulterior del producto finito. Por ejemplo, para aprender la programación es mejor iniciar con un lenguaje de alto nivel del paradigma imperativo de tipo C o PASCAL. En el caso de que el destinatario del programa utilizara el sistema operativo de plataforma móvil con sistema MAC OS, las herramientas para desarrollar aplicaciones imponen usar el *framework* COCOA o XCode y el lenguaje de programación Objective C.

También es posible que al interior de un programa sean introducidas algunas otras funciones de diferente naturaleza, las cuales son escritas en otros lenguajes de programación o en fragmentos de códigos de otro lenguaje (por lo general, en un lenguaje declarativo de interrogación de base de datos). En un proyecto de desarrollo de programa, se elige al menos un lenguaje de programación, pero resulta técnicamente posible elegir otro u otros lenguajes.

Transformación de un programa

Un programa de usuario recorre el siguiente camino hasta su ejecución:

- **Edición**

  Con un editor de texto se escribe el programa en el lenguaje elegido.

- **Compilación**

  En lenguaje de alto nivel, el código fuente se transforma en instrucciones para la máquina (código objeto o código ejecutable).

- **Enlazado**

  Un ejecutable se construye con códigos objeto (uno o más) y librerías de funciones, entre otros.

El resultado de este proceso es un código ejecutable directo para la máquina.

Pero también existe el modo interpretación de ejecución, en el cual cada frase, instrucción, orden o consulta, escritos en código fuente, se transforma, poco a poco, en órdenes, ya sea directamente por el procesador, por otro software o por la máquina abstracta. Este es el caso del intérprete del lenguaje PROLOG, del Shell y del motor de resolución de consultas (SQL, por las bases de datos). En el mismo caso también se encuentra el lenguaje Java en modo interpretado, en donde el código transformado (clases o archivos) es interpretado por la "Máquina Virtual Java".

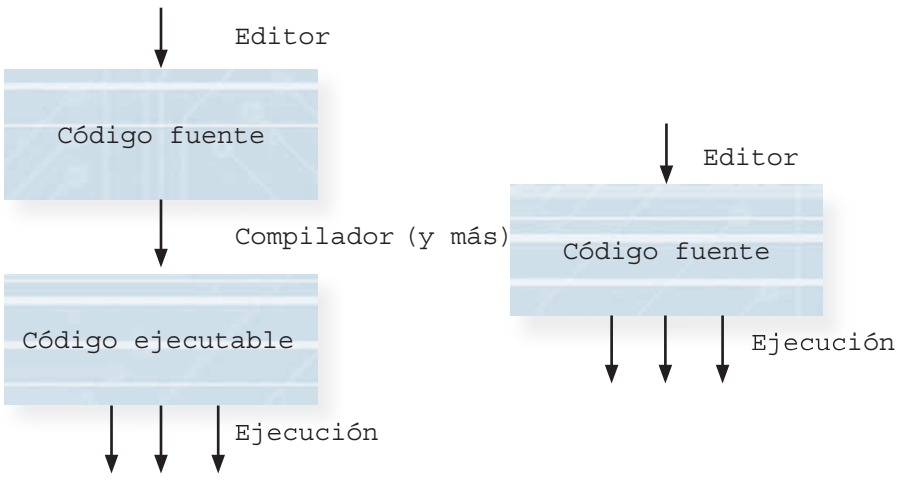


Figura 1.4

En gran parte de los casos, el compilador o el intérprete realiza algunas transformaciones a los programas (optimización de código, detecciones de fin de programa, paralelización de código, etc.), para obtener un código máquina más rápido o más adaptado a la máquina a la cual está destinado.

Para la mayoría de los lenguajes, hay herramientas completas que permiten, en ambientes amigables, la edición y la realización de todos los pasos hasta la construcción del ejecutable de una manera implícita.

Es muy probable que un programa que se compila y se ejecuta por primera vez tenga errores de compilación. También es probable que, después de un tiempo de ejecución, el programa tenga errores lógicos de ejecución; en este caso, se regresa a la edición del código fuente inicial, con el fin de corregir los errores, y luego se desarrollan las otras etapas, hasta la construcción del ejecutable (véase figura 1.5).

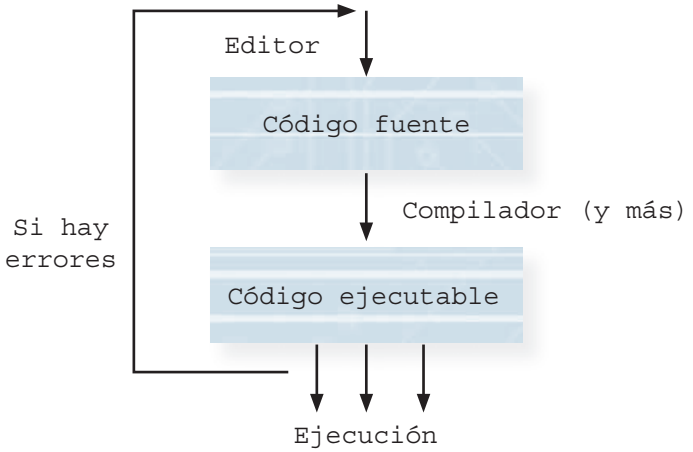


Figura 1.5

A lo largo de este capítulo se presentan el pseudocódigo y los diagramas de flujo como herramientas para el diseño de los algoritmos. Por su parte, el lenguaje C, se aborda con amplitud más adelante en otros capítulos, ya que se trata de un lenguaje imperativo y estructurado, considerado un lenguaje de alto nivel.

1.2 Variables, tipos y expresiones

El objetivo general de un programa es transformar datos en resultados útiles para el usuario. Los datos están almacenados en la memoria principal o en la memoria secundaria, ya sea de manera temporal (durante toda la ejecución del programa o durante una parte del tiempo de ejecución) o de manera permanente. En la mayoría de los lenguajes de programación, los datos son de diferentes tipos, aparecen en expresiones o en las llamadas de funciones y se manejan a través del uso de variables.

Variables

El formato de representación y de estructuración de los datos depende del paradigma del lenguaje de programación y de la opción que el programador ha elegido para representar los datos. En el paradigma imperativo y en el caso de algunos otros paradigmas (por ejemplo, lenguaje PROLOG) existe una noción básica común para el manejo de los datos: la **noción de variable**. La ventaja de las variables es que almacenan datos de entrada, de salida o intermedios. No obstante, existen lenguajes de tipo SQL o XPath que no implementan la noción de variable.

Por lo general, en cada programa aparece al menos una **variable**, lo que significa que en cada programa hay una zona de memoria con un tamaño fijo que contiene un valor de tipo preciso; por ejemplo, un entero representado en forma binaria de Ca2[5] sobre 4 bytes, o una cadena de caracteres de un tamaño máximo de 255.

[5] Véase el apéndice 1 en el CD-ROM.

Cada variable debe tener:

- Un tamaño de memoria ocupada y un modo de representación interna. Por ejemplo, un punto flotante simple precisión sobre 4 bytes o cadenas de caracteres de 100 + 1 caracteres.
- Un conjunto de operadores y de tratamientos específicos que pueden aplicarse a la variable. Si las variables son, por ejemplo, de tipo lógico, se aplican operadores lógicos; pero, si las variables son numéricas, se aplican operadores de cálculo numérico (suma, producto, entre otros).

El nombre de una variable debe ser único y no ambiguo. La unicidad del nombre de la variable durante su ciclo de vida, asegura una semántica correcta de las operaciones (expresiones, órdenes o proposiciones) que implican a la variable. De esta forma, el nombre de una variable es un **identificador** diferente de cualquier palabra clave utilizada en el lenguaje o nombre de una función externa. Generalmente, los nombres de las variables inician con una letra y son sucesiones de letras y cifras y el símbolo _ (guión bajo). Para la cualidad del programa, es preferible que el nombre de una variable sea sugestivo al tratamiento y de un largo de tamaño aceptable, ya que un nombre de variable muy largo puede generar errores de tecleo al momento de la edición del programa, lo que produce pérdidas de tiempo para su corrección. En la determinación del nombre de la variable, también se sugiere utilizar únicamente letras sin acento, para una mejor portabilidad del código o porque la sintaxis del lenguaje no lo permite.

Algunos ejemplos de nombres de variables son los siguientes: `a`, `a1`, `area`, `suma`. También lo son: `a1b159` y `a2b158`; sin embargo, la lectura de un programa con nombres de este tipo sería difícil. Por lo que respecta a la extensión del nombre, una variable llamada `nueva_suma_valores_quantidades`, tomaría mucho más tiempo escribirla.

Se considera que variables de nombre *i*, *j*, *k*, indican variables enteras usadas para índices; en tanto, las variables de nombre *a*, *b* y *c*, por lo general se utilizan para valores numéricos reales (punto flotante); las variables llamadas *p* y *q* se emplean para apuntadores; las variables llamadas *n* y *m* son variables que contienen valores de tamaños de arreglos.

No es obligatorio que la **variable tenga un valor** al saberse que la zona de memoria dedicada a la variable sea ocupada. En algunos momentos, es posible que la variable no tenga ningún valor; en estos casos, se dice que la variable es no-inicializada (por ejemplo, lenguaje PASCAL) o libre (por ejemplo, lenguaje PROLOG). Si la variable posee un valor en un instante T del programa, dicho valor solo es único para ese instante T. A lo largo de la vida de la variable, el valor que tenga esta puede cambiar; la única condición es que los valores guardados sean del mismo tipo de la variable. El cambio de valor de la variable se hace alrededor de una operación explícita de asignación o por efecto secundario,[6] como el cálculo de una función o el tratamiento de un recurso externo de tipo archivo.

Las variables son de varios tipos; en la mayoría de los lenguajes de programación imperativa predominan los siguientes tipos:

- **Variables simples.** Son propias de los tipos básicos, para los datos enteros, flotantes, caracteres y lógicos (pero no en el lenguaje C).
- **Variables compuestas.** La definición del tipo de una variable compuesta depende de la sintaxis del lenguaje de programación y de su poder semántico.
- **Arreglos de variables de tipo simple o tipo compuesto.** Los arreglos sirven para almacenar una sucesión de valores del tipo indicado.

En la mayoría de los lenguajes en los que cada variable tiene una **declaración**, se indica el nombre y el tipo. En ocasiones, también se indica si la variable es estática o dinámica o si el acceso al contenido de la variable es público o privado (por lo general, en lenguajes orientados a objetos). Si la semántica del lenguaje impone la declaración de cualquier variable que se usa, la ausencia de dicha declaración genera un error de compilación. Del mismo modo, si existen variables que están declaradas, pero que no se utilizan, la mayoría de los compiladores envían mensajes explícitos de advertencia, que no son errores, pero sí informaciones realizadas por el programador. También existen casos de lenguajes en los que ninguna variable se declara; en estos lenguajes, a cada aparición de la variable se considera que dicha variable tiene un tipo por defecto. Este es el caso del lenguaje PROLOG, en el cual únicamente una variable sirve para la evaluación de expresiones lógicas o del lenguaje M. También hay lenguajes

[6] Efecto secundario (*side effect* en inglés).

en los cuales si una variable se usa sin definición, se considera que es una variable simple de un tipo indicado por su nombre (por ejemplo, el lenguaje FORTRAN). La sintaxis de las declaraciones de variables es diferente de un lenguaje a otro; sin embargo, un elemento común es que en todos los casos se indican el tipo de la variable y su nombre.

Ejemplos

`integer A, I;` *significa dos variables de nombre A e I y de tipo entero.*

`double Rayo;` *significa una variable de nombre Rayo y de tipo punto flotante doble precisión.*

La noción de variable es generalmente la misma para la mayoría de los lenguajes de programación de tipo imperativo; no obstante, de un lenguaje a otro, o de una computadora a otra, la implementación puede ser diferente. Por ejemplo, según la computadora, un tipo entero se implementa con un tipo de representación binaria y sobre 2, 4 u 8 bytes. Así, en el lenguaje M, del software MATLAB, todas las variables tienen el tipo de doble precisión (64 bits).

Una **variable simple** tiene un nombre único y posee un solo valor de tipo elemental; dicho tipo está declarado explícita o implícitamente. En el ejemplo anterior, las tres variables son simples. A una variable le corresponde una zona de memoria que contiene el valor, donde escribiendo el nombre de la variable se accede a su valor.

Ejemplo

Si después de las declaraciones precedentes se escribe A+I, esto representa una expresión aritmética que usa los valores de las variables A e I.

Un **arreglo** es una variable que tiene un nombre y posee un cierto número de valores del mismo tipo (simple o compuesto), los cuales se encuentran almacenados, uno después del otro, en una zona de memoria contigua. El tipo de cada valor del arreglo también es implícito o explícito. En la declaración de un arreglo, más que el tipo de los elementos del arreglo y el nombre de este, se indica la dimensión.

Ejemplo

`integer YX[10];` *significa un arreglo que contiene 10 valores enteros.*

Según los lenguajes de programación, se puede trabajar o no con todo el arreglo en un solo comando o expresión, o (el caso más común) trabajar con un solo valor del arreglo a la vez.

Ejemplo

Si se trabaja con el lenguaje M, `sum(YX)` *significa la suma de todos los valores y* `5*YX` *significa un arreglo temporario que contiene los valores del arreglo* `YX` *multiplicados por* `5`*. En el lenguaje C, estas expresiones no significan nada, a menos que el usuario defina una función especial sum(...) capaz de tratar arreglos de tipo entero.*

Un valor que compone el arreglo se llama elemento. Un elemento se identifica con el nombre del arreglo y con su posición al interior del arreglo, llamada índice.

Ejemplo

Por la declaración precedente, `YX[1]` *es el elemento de índice 1 del arreglo* `YX`*; este valor se puede utilizar en cualquier expresión aritmética o instrucción.*

Si tomamos en cuenta la definición de una variable entera I, entonces `YX[I]` *es el elemento con el índice del valor de la variable I del arreglo* `YX`.

Según los lenguajes de programación, los índices de un arreglo empiezan en 1 (lenguaje M o FORTRAN o PASCAL), o en 0 (lenguaje C o Java).

La discusión sobre las variables compuestas y los apuntadores está muy extendida y es muy dependiente del lenguaje de programación. En el caso del lenguaje C, que es el que se va a presentar en este texto, se trata con detalle en

los capítulos 2 y 3. En tanto, en el siguiente apartado se estudian los diferentes tipos de variables; normalmente, para cualquier tipo conocido por el programa se puede definir una variable.

El ciclo de vida de una variable inicia en el momento de la **asignación de la zona de memoria**, conforme a su definición. La asignación es realizada por el compilador, si la variable es global (es decir, si el lenguaje es compilado), o por el intérprete, durante la ejecución del programa, si la variable es local o dinámica. El tamaño de la zona de memoria asignada es, por lo general, el tamaño del tipo por las variables simples o el producto del tamaño del tipo del nombre de los elementos (es decir, la dimensión) del arreglo. En el caso de algunos lenguajes de programación que manejan colecciones de datos, el tamaño de memoria asignado es variable. Sin embargo, la reserva de memoria puede ser fija o variable, según el lenguaje de programación, el tipo de cálculo que se hace o el ambiente de ejecución.

Si la variable es local o aparece en una parte de código que termina o si el programador lo indica (con una función `free(X)` en el lenguaje C, por ejemplo), se hace la liberación de la zona de memoria ocupada por la variable.

Una **variable** es **local** si su contenido es accesible únicamente en una parte del programa (por ejemplo, un bloque en el lenguaje C, la resolución de un predicado en PROLOG o el cuerpo de una función que se ejecuta en la mayoría de los lenguajes). En sentido opuesto, también hay variables globales; así, una **variable global** es visible desde cualquier lugar del programa.

Ejemplo

```
#include<stdlib.h>
#include<stdio.h>
int a;
void funcion_impresion(int x)
{

  int b;
  b = sizeof(int);
  printf("Se necesita %d bytes para guardar el valor %d.\n", b, x);
  printf("Se necesita %d bytes para guardar el valor %d.\n", (int)sizeof(x+a),
  x+a);
}

int main()
{
  int b;
  b = 15;
  a = 25;
  funcion_impresion(b);
}
```

Este programa en C tiene una variable global que es visible desde las dos funciones: main y funcion_impresion*. También, hay dos variables locales con el mismo nombre,* b*, en cada una de las dos funciones; cada una de estas variables tiene un contenido diferente.*

Las **variables locales** pueden clasificarse en **estáticas** o **dinámicas**, pero esta clasificación únicamente aplica en algunos lenguajes de programación, por lo que su semántica es diferente de un lenguaje a otro, para indicar el modo de asignación de memoria. La noción se usa en el caso de funciones recursivas, por las cuales las variables estáticas son únicas para todas las llamadas de una misma función. Una **variable estática** tiene su espacio de memoria asignado fuera de las variables dinámicas en lenguajes como C, C++ o VisualBasic. En el ejemplo anterior todas las variables son dinámicas.

Toda vez que una variable tiene su espacio de memoria asignado, su contenido puede ser consultado en lectura o en lectura escritura.

La lectura del contenido de una variable se hace a cada aparición del nombre de la variable, pero si la variable no contiene nada (es decir, no fue inicializada), su lectura produce un error.

En la mayoría de los casos y de los lenguajes de programación (excepto en algunos lenguajes del paradigma declarativo), la escritura (cambio) de un contenido se hace con un operador de **asignación de valor**. Dicho operador de asignación tiene una aridad de dos y generalmente se expresa con la sintaxis siguiente:

```
variable operador_asignacion expresión
```

El operador de asignación cambia según el lenguaje de programación del que se trate; así, es = para los lenguajes C, C++ o Java, e: = para los lenguajes PASCAL o SET y algunos lenguajes declarativos (por ejemplo, LISP, SCHEME, XSLT).

En el operador de asignación, la parte izquierda (el primer operando) constituye la variable y la parte derecha es una expresión del mismo tipo o un tipo compatible por el cual el valor sería convertido al tipo de la variable, si la conversión es posible. Sin embargo, el funcionamiento es siempre el mismo: primero se evalúa la expresión y luego se hace la escritura del valor obtenido en la zona de memoria asignada por la variable.

La expresión que aparece en una asignación debe ser correcta sintáctica y semánticamente (escritura correcta y uso correcto del tipo). Pero, esta verificación de corrección no es una garantía de que la expresión esté correcta al momento de la ejecución de la asignación.

Los **errores de evaluación** pueden aparecer como la división con cero, un valor de índice que está fuera del rango permitido o una operación aritmética que se hace con desbordamiento aritmético. Según el lenguaje del que se trate, si el cálculo de la expresión se hace con errores o excepciones, es posible integrar un código general de tratamiento de la excepción o un código particular (lenguajes ADA, Java o Smalltalk). También es posible que el compilador que introduce verificaciones de corrección, paso a paso, durante la evaluación de la expresión de la parte derecha, señale explícitamente la causa del error, como en el lenguaje Java. Pero estos lenguajes también son considerados lenguajes sin ninguna verificación de este tipo y los errores de cálculo de la expresión pueden ser fatales, como la división con 0, o el cálculo continúa con un valor incorrecto. En el lenguaje C, si el error es fatal, la asignación se interrumpe y el programa también.

Tipos

Un **tipo informático** es el atributo de cualquier dato (constante, variable o dato almacenado en la memoria interna o en la memoria externa) guardado por el programa de manera implícita o explícita. Por lo general, el tipo indica la forma física del contenido del dato. Así, un tipo induce naturalmente una representación interna de los datos; entonces, el tamaño también induce en la semántica del lenguaje un conjunto de operadores que se aplican a los valores pertenecientes a este tipo.

Los tipos son características de los lenguajes de programación y se clasifican en:

- Tipos predefinidos.
- Tipos definidos por el usuario.

Un **tipo predefinido** es un tipo propuesto por el lenguaje con una semántica explícita y un conjunto preciso de operadores. Por su parte, un **tipo predefinido** puede ser:

- Un **tipo básico,** el cual traduce tipos de representación interna de los datos en lenguaje de programación, como enteros, reales (con representación en punto flotante), lógicos, carácter (código ASCII, Unicode, entre otros) y cadena de caracteres (menos frecuente).
- Un **tipo complejo,** el cual traduce un tipo abstracto de datos[7] o un tipo de datos que responde a una necesidad en el paradigma de programación; por ejemplo, el tipo enumerado, el semáforo (en programación concurrente), el mensaje en la programación distribuida asíncrona, etcétera.

[7] Un tipo abstracto es un tipo de datos concebido de manera teórica explicitando la semántica del tipo: cómo funciona y cuáles son las operaciones con este tipo. Por ejemplo, podemos definir un tipo abstracto para modelar la noción matemática de conjunto. El tipo abstracto conjunto tiene definidos los operadores entre conjuntos (reunión, intersección y diferencia) y el operador de pertenecia. Un tipo abstracto se implementa después en su lenguaje de programación.

Ejemplo

En el lenguaje C, los tipos básicos predefinidos son enteros o reales: `char`, `short`, `int`, `long`, `float`, `double`.

Un carácter se asimila como un entero representado en 1 byte, y los valores lógicos se consideran por interpretación de los valores que adquiere; es decir, cualquier valor diferente de 0 es verdad, ya que el valor 0 es falso. Las cadenas de caracteres se conciben como arreglos de caracteres con un carácter especial al final. Los tipos complejos parecen definidos en las librerías estándares: `FILE*`, *para trabajar con archivos;* `clock` *y* `time`, *para trabajar con el tiempo del sistema o absoluto;* `socket` *para trabajar con los sockets, etcétera.*

Muchos lenguajes permiten al programador la definición de sus propios tipos, los cuales son más cercanos al problema que se pretende resolver. El tipo compuesto es un conjunto ordenado de variables con tipos conocidos (`struct` en C, record en PASCAL o PL/SQL).

En el paradigma de la programación orientada a objetos, también hay nociones de tipo jerárquico y de tipo opaco con respecto a la visibilidad o la herencia.

La comprobación de tipificación constituye la operación de verificación de compatibilidad de los tipos al interior de una expresión. La tipificación puede ser de dos tipos:

- **Estática.** Hecha al momento de la compilación.
- **Dinámica.** Hecha al momento de la ejecución del programa.

Esta fase de comprobación de los tipos es necesaria para garantizar la corrección del código y evitar los errores de desbordamiento. Cuando se hace una operación entre tipos diferentes, antes es posible hacer una conversión de un tipo a otro (por lo general, el más débil se convierte en el más fuerte), y luego se realiza la operación. La compatibilidad de los tipos está indicada en la parte de la semántica del lenguaje. La tipificación es fuerte cuando solo son aceptadas las transformaciones para el tipo más fuerte (por ejemplo, el lenguaje PASCAL); en caso contrario, la tipificación se considera débil.

Apuntadores

Un **puntero** o **apuntador** es una variable capaz de referenciar una variable del programa o una dirección de memoria. Este se define como una variable, pero nada más se indica que el tipo es una referencia (a veces una ubicación) de un tipo conocido (estándar o definido por el programador) o de cualquier otro tipo. Este modo de acceso a un contenido, pasando primero por su dirección (ubicación de memoria), permite realizar tratamientos por los cuales los operandos no son conocidos completamente al momento de la ejecución del programa.

No todos los lenguajes implementan esta noción; por ejemplo, M (del MATLAB), R o FORTRAN. La semántica y el uso de apuntadores son muy diferentes de un lenguaje a otro.

Los lenguajes ensambladores implementan esta noción de manera natural, con el modo de direccionamiento indirecto por medio de un registro.

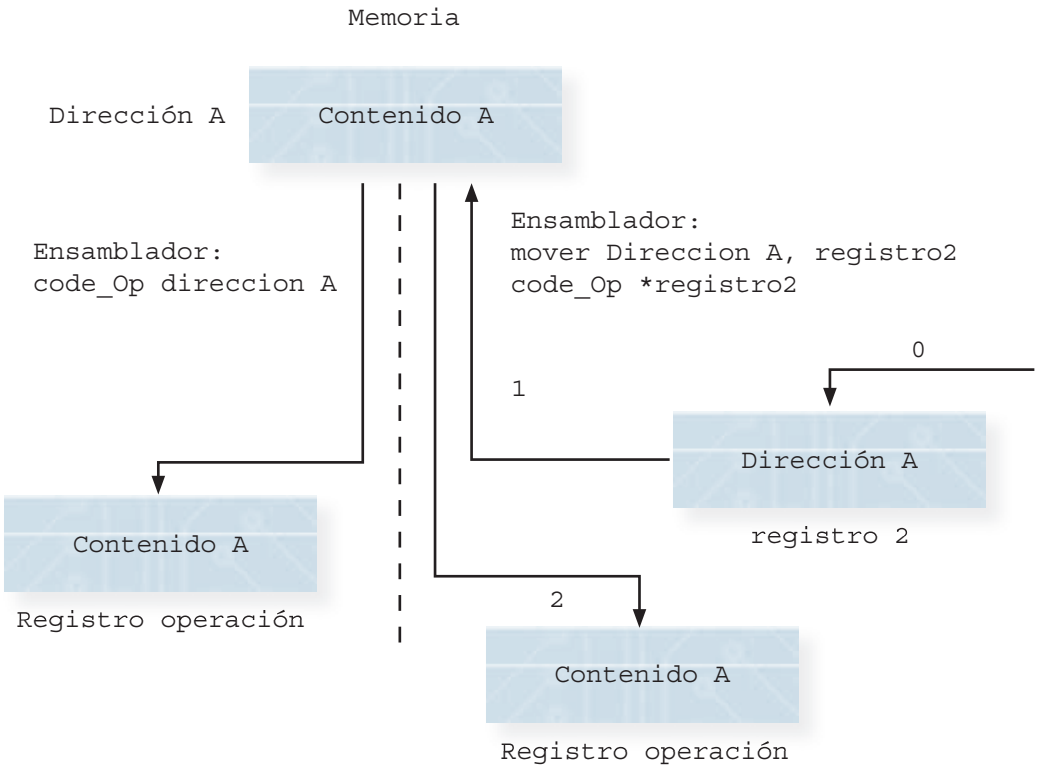
Ejemplo

A continuación se presenta un ejemplo de uso de apuntadores en C y en PASCAL, por el cual el contenido de una variable es accesible y se modifica usando un apuntador sin hacer referencia al nombre de la variable:

```
#include<stdlib.h>
#include<stdio.h>
int main()
{
int a = 12, b = 5;
int *p;
if (a > b) p = &a;
```

```
else
    p = &b;
printf(" El contenido inicial de mi variable preferida :  %d\n", *p);
  a = a + 67;
printf(" El contenido final de mi variable preferida :  %d\n", *p);
  *p = 100;
printf("El valor de a está ahora : %d.\n", a);
}
```

```
PROGRAM codigo_apuntador_ejemplo;
VAR
a,b : integer;
  p : ^integer;
BEGIN
   a := 12;
   b := 5;
if a > b then p := @a
else p := @b;
writeln('El contenido inicial de mi variable preferida :',p^);
  a := a + 67;
writeln('El contenido final de mi variable preferida :',p^);
  p^ = 100;
writeln('El valor de a está ahora ::',a);;
END.
```



Acceder al contenido de la variable A por una operación `Op` usando el acceso directo y el acceso indirecto

Figura 1.6

Expresiones

En programación, una **expresión** es la traducción en lenguaje informático de un cálculo aritmético, lógico o de otra naturaleza. La noción de la expresión fue inspirada de la noción de expresión matemática, por lo que su semántica es similar: la evaluación de una expresión se hace tomando en cuenta los valores que intervienen y aplicando los operadores. En las expresiones, los operadores tienen un orden de evaluación y prioridades. Una expresión contiene, entonces:

- Valores constantes
- Variables
- Operadores
- Paréntesis

La escritura de una expresión en un código implica la evaluación de esta al momento de la ejecución del código. La evaluación se hace tomando en cuenta la prioridad de los operadores.

Los operadores están definidos por la sintaxis del lenguaje, al tiempo que la parte de semántica indica el tipo de los operandos y el tipo del resultado. Por lo general, los operadores del lenguaje de programación son de aridad 1 (un solo operando) o de aridad 2 (la mayoría); el caso de aridad superior a 2 (es decir, de 3 o más operandos) es menos común. La mayoría de los lenguajes de programación usan la forma de infijo para la escritura de las expresiones, que es la escritura en el orden siguiente:

```
operador_aridad_1 operando
operando1 operador_aridad_2 operando2
```

Ejemplo

La expresión matemática $\frac{1}{2} mv^2 + mhg$ *tiene como posible árbol de evaluación el siguiente (la operación de multiplicación es asociativa, entonces hay varias maneras de hacer el cálculo):*

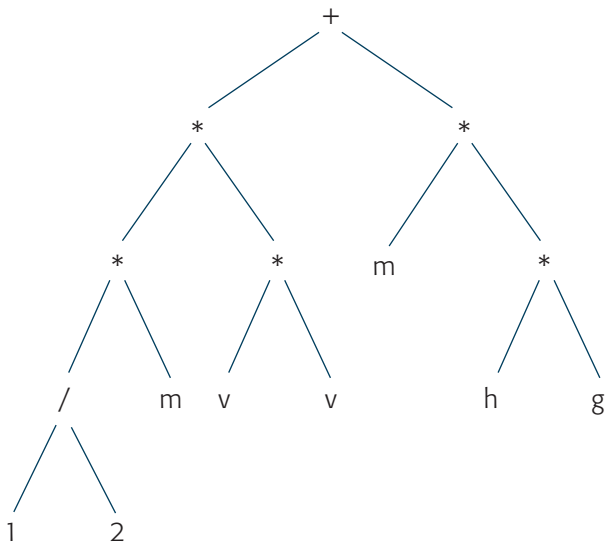


Figura 1.7 Árbol de evolución de la expresión.

*Esta expresión se escribe 1/2*m*v*v+m*h*g en la forma de infijo. En el lenguaje LISP se usa la forma polaca (o forma de prefijo): (* (/ 1 2) m v v) (* m h g)).*

Por los tipos numéricos, se usan las cuatro operaciones aritmética conocidas:

- Suma (adición) +
- Diferencia (sustracción o resta) -
- Producto (multiplicación) *
- División /

En algunos lenguajes hay un operador por el resto de la división entera (el módulo), % (en el lenguaje C), o un operador para la potencia, ^ (en el lenguaje BASIC).

Por tipos que no son numéricos y según el lenguaje, también hay operadores; por ejemplo, por las cadenas de caracteres (si el lenguaje se considera cadena de caracteres como un tipo básico) hay un operador de concatenación (unir) para dos cadenas: + en el lenguaje C++ o | en el lenguaje SQL.

Según la semántica de cada lenguaje de programación, los operadores que corresponden a operaciones aritméticas / lógicas o de transformación se aplican a operandos:

- **de tipos similares o compatibles**, obteniendo un resultado del mismo tipo (por ejemplo, las operaciones aritméticas se hacen entre elementos de tipo numérico) o de otro tipo (por ejemplo, el tipo lógico).
- **de tipos diferentes**; por ejemplo, en el lenguaje C, la adición y la sustracción de un apuntador y de un entero; el resultado significa un nuevo apuntador para la dirección calculada, según el apuntador y el segundo operando.

Otra clase de operadores son los **operadores de orden**, que sirven para comparar el orden de dos valores de tipos numéricos, con el fin de regresar un valor de tipo lógico. En matemáticas, los operadores de orden más comunes son: $<$, $\leq$, $>$, $\geq$, $=$, $\neq$. En la mayoría de los lenguajes, estos operadores se traducen en programas con los siguientes símbolos: <, <=, >, >=, = y <> o !=.

Asimismo, en la mayoría de los lenguajes de programación, también se implementan los operadores lógicos de la lógica de primer orden: la negación (operación de aridad 1), la conjunción y la disyunción (operaciones de aridad 2). Estos operandos lógicos corresponden a las palabras "no", "y", "o". En muchos de los lenguajes, corresponden a los operadores NOT, AND y OR. Las tablas de verdad de estos operadores son:

p	NOT p
V	F
F	V

p	q	p AND q
V	V	V
V	F	F
F	V	F
F	F	F

p	Q	p OR q
V	V	V
V	F	V
F	V	V
F	F	F

En programación, los operadores aritméticos tienen la misma prioridad que en matemáticas; así, las operaciones de * y / tienen la misma prioridad alta, que las operaciones de + y -. En las operaciones aritméticas, los operadores tienen

una prioridad mayor que los operadores de orden; en tanto, los operadores lógicos tienen una prioridad más baja que los otros.

Por ejemplo, si las variables *a*, *b* y *c* tienen valores numéricos, para verificar que *a*, *b* y *c* pueden ser las aristas de un triángulo, en lenguaje matemático se impone que *a*, *b* y *c* serían valores positivos y que cada número verifica la siguiente desigualdad triangular: `x + y < z`.

En lenguaje de programación, estas seis condiciones lógicas que deben cumplirse se escriben con la expresión:

```
a>0 AND b>0 AND c>0 AND a<b+c AND b<a+c AND c<a+b
```

En la expresión anterior no son necesarios los paréntesis, sino que únicamente se utilizan para dar mayor claridad, por lo que cada operación de orden se puede escribir entre paréntesis, así:

```
(a>0) AND (b>0) AND (c>0) AND (a<b+c) AND (b<a+c) AND (c<a+b)
```

Funciones

Se considera que una **función** es una parte de código capaz de realizar una tarea y/o de transformar valores para obtener otro valor. Una función se define por:

- El **nombre**. Este no debe ser ambigüo; según el lenguaje, el nombre debe ser único con respecto a las variables globales y a otras funciones.
- El tipo de **valor** que la función regresa.
- El número fijo (o variable) de **parámetros** y la lista ordenada de **tipo** aceptable por los parámetros.
- El **código**. Este es único para cada función.

Cercanas a la noción de función (o idénticas por el lenguaje C), se encuentran las nociones de procedimiento, rutina o subrutina, las cuales significan una parte del programa encargada de realizar una tarea sin regresar expresamente un valor. Los valores calculados o transformados por el código se regresan en la lista de los parámetros.

Si el lenguaje permite la redefinición de las funciones, por ejemplo, los lenguajes orientados a objetos, como C++ o Java, o que las funciones tengan varias listas de parámetros, el código de una función no es único. En el primer caso se toma en cuenta la última definición de la función, mientras que en el segundo se hace la correspondencia entre la lista de parámetros actuales y las listas de parámetros.

Una vez que la función está definida (y si no tiene restricciones de acceso; por ejemplo, no es privada, como en el caso del lenguaje Java, o no es una función interna de otra función, como en el lenguaje C), en todo el código se pueden hacer una o varias llamadas a la función, con la única restricción de que la lista de los **parámetros reales** (especificados en la llamada) correspondan en nombre y tipo con los **parámetros formales** (que aparecen en la definición de la función).

La llamada de la función se hace especificando:

- El nombre de la función.
- La lista de los parámetros reales, los cuales pueden ser expresiones que se evalúan o variables.

Por su parte, la sintaxis de una llamada de función es casi la misma para prácticamente todos los lenguajes (a excepción de algunos lenguajes funcionales, como LISP o SCHEME) y es inspirada en la notación matemática:

```
Nombre_función(parametro1, parametro2, …)
```

La sintaxis de la definición de una función varía considerablemente de un lenguaje a otro, al igual que la semántica de la definición y el modo de ejecución de las llamadas. En el caso de las llamadas de funciones, estas tienen varias semánticas, según el paradigma del lenguaje de programación. Por el paradigma imperativo y por las funciones que regresan valores, las llamadas se comportan como expresiones del tipo regresado.

De acuerdo con el lenguaje de programación, los parámetros pueden modificarse o no en el código de la función, donde el valor del parámetro a la salida de la función es cambiado. O se indica expresamente si los **parámetros** son **de entrada** (cuando sus valores no cambian) o **de salida** (por ejemplo, el lenguaje PL/SQL) si los valores van a cambiar.

Un parámetro real que no es de salida puede ser cualquier expresión posible del parámetro formal.

En el lenguaje C solo existe la noción de parámetro de entrada y de salida, lo cual depende de la forma en que se transmite: el valor indicado por una variable o una expresión; o un apuntador al contenido de una variable. Solo un apuntador transmitido como parámetro puede cambiar el contenido de la memoria. Hablamos de parámetros transmitidos por valor o por referencia.

Ejemplo

Una función que calcula la suma de los valores de dos elementos o de una lista de elementos.

Por la suma de dos elementos, implantamos las funciones en los lenguajes C, PASCAL y PL/SQL, y por la suma de una lista realizamos las implementaciones en los lenguajes PROLOG y LISP.

El concepto de lista no tiene un tipo predefinido en los tres lenguajes antes mencionados, por lo que es muy diferente en PROLOG y en LISP. Por su parte, en el lenguaje PROLOG no existe la noción de función regresando cualquier tipo de valor, sino que las funciones (llamadas predicados) regresan valores de verdad.

Definición de la función suma

- **Lenguaje C – dos versiones**

```
int sumaC1(int a, int b)
{
return a + b;
}

void sumaC2(int a, int b, int *valor)
{
  *valor = a + b;
}
```

- **Lenguaje PASCAL**

```
FUNCTION  SUMA2(a : integer; b : INTE-
GER) : INTEGER;
BEGIN
  SUMA2 := a+b;
END;
```

- **Lenguaje PL/SQL**

```
CREATE OR REPLACE PROCEDURE SUMA2(a IN
integer, b in INTEGER, s OUT INTEGER)
BEGIN
  s := a+b;
END;
```

- **Lenguaje PROLOG**

```
suma(0, []).
suma(X, [X]).
suma(S,[X|L]):-suma(Y,L),S is X +Y.
```

Código de las llamadas

- **Lenguaje C**

```
int suma;
printf("suma1:%d\n",sumaC1(12,56));
  sumaC2(12,56, &suma);
printf("suma 2: %d\n", suma);
```

- **Lenguaje PASCAL**

```
VAR valor_suma : integer;
BEGIN
valor_suma := SUMA2(12, 67);
  WRITELN('La suma es:', valor_suma);
END.
```

- **Lenguaje PL/SQL**

```
DECLARE
  VAR VALOR_SUMA INTEGER;
BEGIN
  SUMA2(12, 67, VALOR_SUMA);
END.
```

- **Lenguaje PROLOG**

```
?- suma(8, [1, 2, 4]).
false.
?- suma(XX, [11, 2, 45]).
XX = 58 .
```

- **Lenguaje LISP**

```
>(suma ())
0
>(suma '(1 2 3 4 5))
15
```

- **Lenguaje LISP**

```
(defun suma (lista)
« Calculo de la suma de dos elemen-
tos de la lista»
(if (null lista) 0
     (+ (first lista) (suma (rest
lista))
     )
))
```

En ocasiones, las funciones tienen efectos de bordo, transformando contenidos en otras zonas de la memoria o cambiando los estados de los dispositivos de entrada/salida. Por ejemplo, una función `clear( )` sin parámetros en lenguaje C o la rutina `ClrScr` en PASCAL, que borra la pantalla de trabajo.

En todos los lenguajes, más que las funciones definidas por el usuario, se utilizan funciones que provienen de librerías externas, estándares (anexadas al compilador o al intérprete) o funciones adicionales. La función citada, que borra la ventana de trabajo, proviene de una librería estándar para los dos lenguajes.

1.3 Pseudocódigo

Un **pseudocódigo** (falso lenguaje) está formado por una serie de palabras con un formalismo muy sencillo, que permite describir el funcionamiento de un programa. Se usa tanto en la fase de diseño como en la fase de análisis.

El pseudocódigo describe un algoritmo utilizando una mezcla de frases en lenguaje común, instrucciones de programación y palabras clave que definen las estructuras básicas. Su objetivo es permitir que el programador se centre en los aspectos lógicos de la solución de un problema.

El pseudocódigo utiliza **expresiones matemáticas**, **expresiones lógicas** y la noción de **variable** (sencilla, arreglo, pila, cola, conjunto, etcétera). El pseudocódigo se puede extender para expresar tipos complejos y operaciones entre variables y constantes de este nuevo tipo.

Nociones básicas: variables, tipos y expresiones

Una **variable** es un contenido de memoria que contiene un valor que podemos cambiar; es decir, que varía. Una variable tiene un **nombre** (fijo y único) y un **valor** (variable durante la ejecución del algoritmo).

Las expresiones matemáticas contienen los operadores conocidos, constantes y funciones matemáticas. Por ejemplo:

$$X - 1, \sqrt{2 + 16 + 18}, \ \text{sen}(2) * \cos(x)$$

Una expresión lógica contiene expresiones matemáticas, operadores de comparación y operadores lógicos.

- Los operadores de comparación son: $=, \neq, >, <, \leq, \geq$.
- Los operadores lógicos son: `AND`, `OR`, `NOT`.

Ejemplo

$$X - 1 = 4, \sqrt{x + 16} \neq 5, \ \text{sen}(x) * \cos(x) \geq 0.2, \ x < y \ \text{AND} \ y < z$$

En este caso, las primeras tres expresiones contienen únicamente un operador de comparación, mientras que la última expresión es la traducción de la expresión matemática: $x < y < z$.

Una variable contiene valor de entrada o de salida (resultados) o cálculos intermediarios.

Para cambiar o dar un valor a una variable, se utiliza una lectura o una asignación. La **lectura** de una variable se realiza de la siguiente forma:

```
Lectura (variable)
```

El funcionamiento de una variable es conforme al siguiente orden:

1. El usuario del código entrega un valor de buen tipo y este valor se guarda en *variable*.

Ejemplos

Lectura (alfa).

Se lee (entrega) un valor por la variable alfa.

Lectura (alfa, beta).

Se leen dos valores que se entregan luego: primero a la variable alfa, segundo a la variable beta.

2. Después de una lectura, y hasta un nuevo cambio de valor, el valor contenido en la memoria para una variable no cambia.

La **asignación** de un valor a una variable se realiza de esta forma:

```
variable ← expresión (matemática o lógica)
```

La expresión (matemática o lógica) puede contener la variable misma, mientras que su tipo es el mismo que el tipo de la variable.

La significación de la asignación se efectúa realizando el cálculo de la expresión y, luego, el contenido de la variable se cambia con el valor de ese cálculo; por tanto, el valor anterior de la *variable* se pierde.

Ejemplos

```
x ← 4 + 22
alfa ← beta x gamma
i ← i + 1
```

En la primera asignación aparece el símbolo matemático π, el cual posee una significación muy precisa, pero en el momento de la traducción del pseudocódigo en un lenguaje de programación, debemos buscar cómo se implementa o se puede implementar esta constante.

Por su parte, la última asignación tiene el siguiente funcionamiento: con el valor actual de la variable i se hace el cálculo de i + l, cuyo valor sería guardado en la variable i. Por ejemplo, si la variable i vale 3, después de esta asignación el contenido de la variable es 4.

La escritura de un contenido se hace de una manera muy simple:

```
Escritura (variable)

Escritura (constante)
```

El funcionamiento es evidente; el valor de la variable o de la constante se pone en la salida del pseudocódigo.

La principal función del pseudocódigo consiste en decir qué debe hacer el código y cómo; por esta razón, la escritura es muy simple, sin indicar el formato de salida, y con un cierto número de cifras después del punto decimal; por ejemplo, con una política de caracteres o determinando en qué lugar de la pantalla se escribe.

Ejemplos

```
Escritura ("Aquí termina el programa")
Escritura (i)
Escritura (j)
```

La primera escritura, "Aquí termina el programa", es una constante de tipo cadena de caracteres.

Las siguientes escrituras se colocan en la salida de los valores de las variables i y j, en ese orden.

Para ser más específico, antes de una lectura o de una escritura, se puede poner una *Escritura* (*mensaje*).

Ejemplos

```
Escritura ("Indica por favor el valor de a:")
Lectura (a)
…
Escritura ("Mi calculo final es:")
Escritura (X)
```

Estructura general del pseudocódigo

Un pseudocódigo se escribe para dar las grandes líneas del cálculo; su objetivo es compartir con los demás programadores su visión de la resolución del problema. Hay dos principios en la escritura de un pseudocódigo:

- Al inicio se escriben todas las variables que se usan en pseudocódigo; cada una con su nombre y su tipo.
- Las líneas del pseudocódigo que siguen son órdenes (instrucciones o estructuras) que se ejecutan de arriba hacia abajo; primero una orden y después otra, así sucesivamente.

El pseudocódigo que se utiliza para la descripción de un algoritmo o para indicar los pasos de resolución de un problema contiene **estructuras de control**, las cuales se utilizan para describir las instrucciones de los algoritmos. Hay cuatro tipos de estructuras:

- Secuencial
- Selectiva
- Iterativa
- Anidamiento

Estructuras componentes del pseudocódigo

Estructura secuencial

La **estructura de control secuencial** tiene la siguiente forma:

```
instrucción1
instrucción2
     …
instrucciónk
```

Las instrucciones se ejecutan en el orden indicado por los índices: de arriba hacia abajo y una después de la otra.

Ejemplo

Primer pseudocódigo

Lectura de x y cálculo del cuadrado de x:

```
Real x, y;
Lectura (x)
 y ← x × x
Escritura (y)
```

Observaciones:

- *La primera línea contiene la declaración de las variables; x y y son las únicas variables y tienen el mismo tipo.*

- *Las instrucciones se ejecutan en el siguiente orden: lectura, asignación, escritura.*
- *No hay ambigüedades en cada instrucción y pueden ejecutarse.*
- *Las variables que se usan tienen un valor correcto al inicio o al fin de cada instrucción y en todo el pseudocódigo.*

La estructura secuencial es la base del pseudocódigo, pero no es suficiente para resolver todos los problemas.

Estructura selectiva

Las **estructuras selectivas** permiten expresar las elecciones que se hacen durante la resolución del problema. Hay varios tipos de estructuras selectivas:

- Selectiva simple.
- Selectiva doble (alternativa).
- Selectiva múltiple.
- Selectiva casos (múltiple).

La **estructura selectiva simple** es de la siguiente forma:

```
si expresión lógica entonces
instrucciones fin si
```

En esta estructura, primero se hace el cálculo de la expresión lógica; si el valor de esta expresión es cierto (no falso) se ejecutan las *instrucciones* (puede ser una sola o más de una). Si el valor de la expresión lógica es falso, no se ejecuta nada.

Las palabras *si, entonces* y *fin si*, son palabras clave que permiten estructurar y dar un sentido a las instrucciones. Por otro lado, es posible escribir la estructura anterior como:

si *expresión lógica* ***entonces***
instrucciones

fin si

Esta escritura no es tan clara. ¿Dónde inicia y dónde termina la estructura *si*?, ¿dónde empiezan las instrucciones que se ejecutan cuando la expresión es cierta? Por estas razones, es mejor que las partes que componen una estructura (en nuestro caso selectiva, aunque también aplica para todas las estructuras selectivas e iterativas) se escriban con algunos espacios y que la parte *si* se alinee con la parte *fin si*.

La **estructura selectiva alternativa** es de esta forma:

```
si expresión lógica entonces
        instrucciones1
            si no
        instrucciones2
            fin si
```

Primero, se hace el cálculo de la expresión lógica. Si el valor de esta expresión es cierto (no falso) se ejecutan las *instrucciones*$_1$. Si no, se ejecutan las *instrucciones*$_2$.

Ejemplo

Verificar si un número entero es o no divisible entre 3.

La entrada es el número n*, la salida es un mensaje de tipo Sí o No.*

```
Integer N
Lectura (N)
resto ← N%3
si resto = 0 entonces
  Escritura ("SÍ")

sino
  Escritura ("NO")

fin si
```

La **estructura selectiva múltiple** es usada para anidar condiciones lógicas mutuamente excluyentes. Su forma es la siguiente:

si *expresión lógica*$_1$ **entonces**
instrucciones$_1$
sino si *expresión lógica*$_2$ **entonces**
instrucciones$_2$
sino si *expresión lógica*$_3$ **entonces**
instrucciones$_3$
…
sino
instrucciones$_n$
fin si

Esta estructura se ejecuta de la siguiente manera:

Se hace el cálculo de la expresión `lógica`$_1$, si el resultado es *cierto* se ejecutan `instrucciones`$_1$ y la instrucción selectiva se termina. Si no, se hace el cálculo de la *expresión* `lógica`$_2$; si el resultado es *cierto* se ejecuten *instrucciones*$_2$ y la instrucción selectiva se termina… Si todas las expresiones lógicas son *falso*, entonces se ejecutan *instrucciones*$_n$.

Ejemplo

Resolver la ecuación de primer grado que tiene su forma matemática más general:

$ax + b = 0$

La entrada está formada por los dos parámetros de la ecuación, a y b, que serán guardados en dos variables de tipo punto flotante. La salida será un mensaje sobre la raíz de la ecuación y, en algún caso, su valor.

Los casos que pueden aparecer y que deben tratarse de manera diferente serán:

- $a = 0$ *y* $b = 0$: *cualquier número real es una solución.*
- $A = 0$ *y* $b \neq 0$: *no hay ninguna solución.*
- $a \neq 0$: *la raíz es única y de valor –a/b.*

Una proposición de pseudocódigo es la siguiente:

```
Real a, b
```

```
Real x
Lectura (a)
Lectura (b)
si a ≠ 0 entonces
  Escritura ("Hay una única raíz")
    x ← b/a
  Escritura (x)
sino si b ≠ 0 entonces
  Escritura ("No hay ninguna raíz")
sino
  Escritura ("Hay una infinidad de raíces")
fin si
```

Este pseudocódigo contiene una estructura selectiva múltiple con condiciones lógicas exclusivas. La resolución se puede hacer con un pseudocódigo que contiene dos estructuras selectivas alternativas, una añadida a la otra:

```
Real a, b, x
Lectura (a)
Lectura (b)
si a = 0 entonces
  si b = 0 entonces
    Escritura ("Hay una infinidad de raíces")
  sino
    Escritura ("No hay ninguna raíz")
  fin si
sino
  Escritura ("Hay una única raíz")
  x ← b/a
  Escritura (x)
fin si
```

Las dos soluciones propuestas son semánticamente equivalentes, por lo que se realiza el mismo tratamiento; depende únicamente de la manera de escribir y leer el código y de que los participantes en el desarrollo de la solución del problema prefieran una u otra.

En la segunda versión, en lugar de dos declaraciones de variables, también tenemos una sola para las tres variables del código; sin embrago, la primera versión permite nada más una separación "lógica" entre las variables de entrada y la única variable de salida.

La **estructura selectiva múltiple-casos** se usa cuando un mismo valor se compara con varios valores. Su forma es la siguiente:

```
seleccionar expresión
caso valor1
instrucciones1
caso valor2
instrucciones2
…
en otro caso
instruccionesn
fin seleccionar
```

La *expresión* puede ser una sola variable. Primero, se obtiene el valor de esta expresión y se compara con cada $valor_i$; si hay expresión = $valor_i$, se ejecutan las $instrucciones_i$. Si ningún valor corresponde, se ejecuta la parte '***en otro caso***', $instrucciones_n$.

Ejemplo

Según el valor de una variable de entrada, se escribe:

- *"FALSO" si el valor de la variable es 0.*
- *"CIERTO" si el valor de la variable es 1.*
- *"INDEFINIDO" si el valor de la variable es −1.*
- *"ERROR" en otros casos.*

El pseudocódigo es muy simple, está estructurado en dos partes: la lectura de la variable y una estructura selectiva de tipo "caso":

```
Integer x
Lectura (x)
seleccionar x
caso 0
  Escritura ("FALSO")
caso 1
  Escritura ("CIERTO")
caso -1
  Escritura ("INDEFINIDO")
en otro caso
  Escritura ("ERROR")
fin seleccionar
```

Observación: *El orden de tratamiento de los casos es libre; se puede poner cualquier permutación de los valores −1, 0 y 1, únicamente al final se pone "en otro caso".*

Estructura iterativa

Las **estructuras iterativas** abren la posibilidad de ejecutar un grupo de instrucciones más de una vez; es decir, sirven para ejecutar varias veces una misma parte de un código. Hay varios tipos de estas:

- Bucle mientras
- Bucle repetir
- Bucle para (cada)

La **estructura iterativa mientras** (`while`) tiene la siguiente forma:

```
mientras expresión lógica hacer
        instrucciones
fin mientras
```

Su ejecución es la siguiente:

Se calcula la *expresión lógica* y, si su valor es cierto, se ejecutan las *instrucciones* y se hace un nuevo cálculo de la *expresión lógica*. Entones, en total, **las *instrucciones* se ejecutan 0 o varias veces**, dependiendo del valor de la *expresión lógica*.

Ejemplo

Buscar el entero m más pequeño, pero que sea mayor que un número real positivo x, con x ≥ 1.

¡Cuidado! Al interior de las instrucciones hay que hacer modificaciones de las variables que componen la *expresión lógica*, porque la expresión siempre sería cierta y la estructura no se terminaría nunca.

La entrada del problema es el número x y la salida es el número entero m. La idea es incrementar en 1 una variable m iniciada con el valor 0 hasta que la variable m sea más grande que la variable de entrada x. El pseudocódigo es:

```
Real x
Integer m
Lectura (x)
m ← 0
mientras m<x hacer
m ← m + 1
fin mientras
Escritura (m)
```

Ejemplo

Buscar el número entero más grande de forma 2^k que sea menor que un número real positivo x, con $x \geq 1$.

La entrada del problema es el número x y la salida es el número $m = 2^k$; asimismo, podemos considerar que la variable k es también una salida.

La resolución de este problema es parecida a la del problema anterior:

Se calcula la potencia más pequeña de $2(2^j)$, que sería más grande que x; después, se hace un paso detrás. El pseudocódigo es de la siguiente forma:

```
Real x
Integer m, k, j
Lectura (x)
m ← 1
j ← 0
mientras m ≤ x hacer
  m ← m*2
  j ← j + 1
fin mientras
m ← m/2
k ← j-1
Escritura ("El numero que se busca es:")
Escritura (m) Escritura ("es la potencia 2 de")
Escritura (k)
```

¡Cuidado! También al interior de esta estructura repetitiva, durante las *instrucciones*, hay que hacer actualizaciones de las variables que componen la *expresión lógica*, de otra manera, el valor de la expresión no cambia y la estructura **repetir** se ejecutaría infinidad de veces.

*En este ejemplo de pseudocódigo, se muestra que antes y después de la estructura repetitiva **mientras**, entre las variables m y j, siempre existe la relación $m = 2^j$.*

La **estructura iterativa repetir** (`repeat`) tiene la siguiente forma:

```
repetir
instrucciones
hasta que expresión lógica
```

Su ejecución es la siguiente:

Se ejecutan las *instrucciones* y se hace el cálculo de la *expresión lógica*. Si su valor es falso, se ejecutan de nuevo las *instrucciones* y se hace un nuevo cálculo de la *expresión lógica*.

En resumen, las *instrucciones* se ejecutan una o más **veces**, dependiendo del valor de la *expresión lógica*.

Una estructura ***repetir*** es equivalente a:

```
        instrucciones
mientras NOT (expresión lógica)
        instrucciones
        fin mientras
```

Ejemplo

Para un número real x entre 0 y 1 (0 <x< 1), una base de numeración b (b ≤ 10) y un número entero positivo k, buscar las k primeras cifras después del punto decimal de la representación de x en base de b.

Las entradas del problema son tres: el número x, la base b y el número de cifras k. Mientras que la salida sería una sucesión de cifras en base b (un número entre 0 y b − 1). De manera más formal:

$$x_{(10)} = 0.c_1c_2\ldots c_{k-1}c_{k(b)}$$

El algoritmo de transformación de un número en base 10 en otra base se aborda en el apéndice 1; la idea es hacer multiplicaciones con la base b, tomando después la parte entera de cada multiplicación. Así, el pseudocódigo:

```
Real x
Integer b, k
Real y, p
Integer i, c
Lectura (x)
Lectura (b)
Lectura (k)
y ← x
i ← 0
repetir
   p ← y*b
   c ← parte_entera(p)
   y ← parte_fraccionaria(p)
   Escritura (c)
   i ← i + 1
hasta que i ≥ k
```

Este pseudocódigo reviste interés por varias razones:

- *Hasta ahora las salidas de los algoritmos se han colocado al final de todos los cálculos; desde que se obtiene una cifra de la representación fraccionaria, esta cifra contenida en la variable c, se escribe y a cada paso se calcula un nuevo valor en la iteración siguiente.*
- *Se ejecutan exactamente k iteraciones; la variable i indica, de manera muy precisa, cuántas iteraciones se ejecutaron.*
- *La manera de trabajar con la variable i no es única, también se puede escribir así:*

```
…
i ← 1
repetir
   p ← y*b
   c ← parte_entera(p)
   y ← parte_fraccionaria(p)
   Escritura (c)
   i ← i + 1
hasta que i > k
```

El sentido de la variable i es ahora: se está ejecutando la iteración número i.

- *Se requiere el cálculo de las partes entera y fraccionaria; entonces, se indica con claridad la "llamada" de funciones, que normalmente están implementadas en casi todos los lenguajes de programación.*
- *La variable p es intermediaria y sirve para hacer una sola vez la multiplicación de y por b. El uso de esta variable no es obligatorio, se puede escribir directamente y en el siguiente orden:*

```
c ← parte_entera(y*b)
y ← parte_fraccionaria(y*b)
```

Nota: *Nada más que la misma multiplicación se hace dos veces.*

- *Otra variable intermediaria es y, que guarda los valores intermediarios por el cálculo de la representación de x; al inicio, esta variable toma el valor de x. Es posible usar directamente x en los cálculos; pero, al final de la estructura repetitiva, el valor de x se desnaturaliza y el valor inicial se pierde.*

Una estructura iterativa que toma en cuenta la noción de variable-contador es la **estructura iterativa para (`for`)**, la cual tiene la siguiente forma:

```
para i de inicio hasta fin [paso p] hacer
    instrucciones
fin para
```

Donde *i* es una variable (simple) e *inicio*, *fin*, *p*, son valores numéricos. Si el paso no es declarado, su valor es 1.

Su ejecución es repetitiva y su funcionamiento es el siguiente:

La *i* recibe el valor *inicio* y se ejecutan las *instrucciones*; luego, *i* se incrementa el valor de *p* (el paso) y se reejecutan las *instrucciones*, si el valor de *i* es menor que *fin*.

En resumen, las *instrucciones* se ejecutan 0 o varias veces, dependiendo de los valores de *inicio*, *fin* y el paso.

El valor de la variable *i* se puede usar al interior de las *instrucciones*, pero no puede ser modificado.

Esta estructura es equivalente a:

```
i ← inicio
mientras i ≤ fin hacer
instrucciones
i ← i + p
fin mientras
```

Ejemplo

Obtener todas las potencias de un número a desde a^1 hasta a^k, donde a y k son valores de entrada; a es un número real y k es un entero positivo.

El uso de una estructura repetitiva "para cada" es evidente:

```
Real a, p
Integer k, i
Lectura (a)
Lectura (k)
p ← 1
para cada i de 1 hasta k
p ← p*a
  Escritura (p)
fin para
```

Es muy fácil verificar que al final de cada iteración la variable p contiene el valor de a^k. El pseudocódigo funciona también por un número k entregado, que es 0 o un valor negativo; en este caso, no se ejecuta nada.

Ejemplo

Obtener la representación fraccionaria de un número. El pseudocódigo del ejemplo anterior se puede escribir con una estructura iterativa "para cada":

```
Real x
Integer b, k
Real y, p
Integer i, c
Lectura (x)
Lectura (b)
Lectura (k)
y ← x
para cada i de 1 hasta k paso 1
p ← y*b
c ← parte_entera(p)
y ← parte_fraccionaria(p)
Escritura (c)
fin para
```

En los últimos dos ejemplos, si se quieren guardar los valores de salida, las potencias de *a* y las cifras en base *b* del uso de las variables simples no son suficientes, por lo que es indispensable usar los arreglos.

Uso de los arreglos

En la sección 1.2 presentamos las nociones de variable en general y de arreglo. Como se vio, un arreglo se caracteriza por un tipo básico (entero, cadena de caracteres, …), su nombre (como cualquier variable) y su tipo.

En el pseudocódigo también es útil poder usar los arreglos. Así, en la parte de la declaración debemos establecer que la variable es un arreglo, indicando su dimensión. Por ejemplo:

```
Integer B, A[10]
Real X [200]
```

Aquí, podemos ver que *B* es una variable simple, *A* es un arreglo de valores enteros y *X* es un arreglo de valores en punto flotante. Los dos arreglos tienen una dimensión fija: 10 por el arreglo *A* y 200 por el arreglo *X*. También, es posible declarar arreglos con una dimensión variable, pero se debe tomar en cuenta, en la escritura de los programas, porque en algunos lenguajes de programación la definición de arreglo es más complicada que una simple declaración. Por ejemplo:

```
Integer N
Integer A[N]
```

En este caso, el arreglo *A* no se puede usar si la variable *N* no está definida; así, es preferible que durante la ejecución del código, el valor de esta variable no cambie.

Ejemplo

La suma de los elementos de un arreglo que se lee de entrada:

```
Integer N, suma, i
Integer A[N]
Lectura (N)
para cada i de 1 hasta N hacer
```

```
   Lectura (A[i])
fin para
/* ahora el arreglo está lleno */
suma ← 0
para cada i de 1 hasta N hacer
  suma ← suma + A[i]
fin para
Escribir (suma)
```

Las partes entre / y */ son comentarios, que ayudan al lector a entender mejor el código.*

El arreglo *A* tiene una dimensión variable. Como un consejo de programación podemos indicar que cuando la dimensión posible no es demasiado grande, conviene hacer la declaración con una constante, a fin de evitar trabajo de programación más difícil.

Ejemplo

El cálculo de las potencias de a, desde a^1 hasta a^k. Sin reducir la calidad de la solución ni su generalidad, podemos suponer que $k \leq 30$. En esta versión, los números de forma a^i serían guardados en un arreglo C[30].

El pseudocódigo es, por tanto:

```
Real a, C[30]
Integer k, i
Lectura (a)
Lectura (k)
p ← 1
para cada i de 1 hasta k
  p ← p*a
  C[i] p
fin para
para cada i de 1 hasta k
 Escritura (C[i])
fin para
```

Funciones y procedimientos

Las funciones y los procedimientos son partes de un programa que van a ser ejecutadas, una o varias veces, con los valores transmitidos en los **parámetros**:

- **Función.** Recibe parámetros (uno o varios) y calcula un **valor de regreso**.
- **Procedimiento.** Recibe parámetros, pero no regresa explícitamente ningún valor.

Los parámetros son de tipos conocidos (simples o arreglos) y pueden ser de entrada o de salida. Un parámetro de entrada sirve para introducir los valores necesarios al cálculo y un parámetro de salida va a cambiar su valor durante la ejecución de la función o del procedimiento, para entregar resultados. La definición de un parámetro es la siguiente:

```
[IN/OUT] tipo_parametro nombre_parametro
```

Con la palabra "IN" se indica un parámetro de entrada y con la palabra "OUT" se indica un parámetro de salida. Los parámetros de entrada siempre deben tener valores; al contrario de los parámetros de salida.

Las funciones y los procedimientos se declaran una sola vez y pueden ejecutarse varias veces. Una ejecución se denomina **llamada**.

La declaración de un procedimiento es:

```
procedimiento nombre_procedimiento (lista de definiciones de parametros)
inicio
…
fin procedimiento
```

La llamada de un procedimiento es de la siguiente forma:

nombre procedimiento(parametro$_1$, parametro$_2$, …)

Donde los *parametros*$_1$ son variables o arreglos, expresiones o constantes. Hay casos en los que los parámetros corresponden en número y tipo con la definición.

Ejemplo

La definición de un procedimiento para la lectura de un arreglo:

```
procedimiento lectura_arreglo (IN Integer dimension, OUT Integer X[])
inicio
  Integer i
  para cada i de 1 hasta dimension hacer
    Lectura (X[i])
  fin para
fin procedimiento
  Ejemplo de dos llamadas:
lectura_arreglo(100, A)
lectura_arreglo(N, X)
```

Una **función** se define como:

```
función tipo nombre_funcion (lista de definiciones de parámetros)
inicio
….
regresa valor
fin función
```

La función tiene que regresar o devolver un valor del mismo tipo de la función. Una llamada de función es parecida a una llamada de procedimiento y puede ser escrita en una expresión.

Ejemplo

La definición de una función para la suma de los elementos de un arreglo es la siguiente:

```
función Integer suma_arreglo(IN Integer dimension, IN Integer X[ ])
inicio
   Integer i, ss
   ss ← 0
   para cada i de 1 hasta dimension hacer
     ss ← ss + X[i]
   fin para
   regresa ss
fin funcion
```

El programa completo para calcular la suma de los elementos de un arreglo puede ser el siguiente:

```
Integer N, suma
Integer A[N]
inicio
   Lectura (N)
   lectura_arreglo (N, A)
   Escribir(suma (N, A))
fin
```

1.4 Diagrama de flujo

Los diagramas de flujo son comunes en varios dominios técnicos y se usan para poner en orden los pasos a seguir o las acciones a realizar. Su principal ventaja es que tienen la capacidad de presentar la información con gran claridad, además de que se necesitan relativamente pocos conocimientos previos para entender los procesos y/o el objeto del modelado. Por ejemplo, en el siguiente diagrama se presentan los pasos a seguir cuando alguien sale de vacaciones:

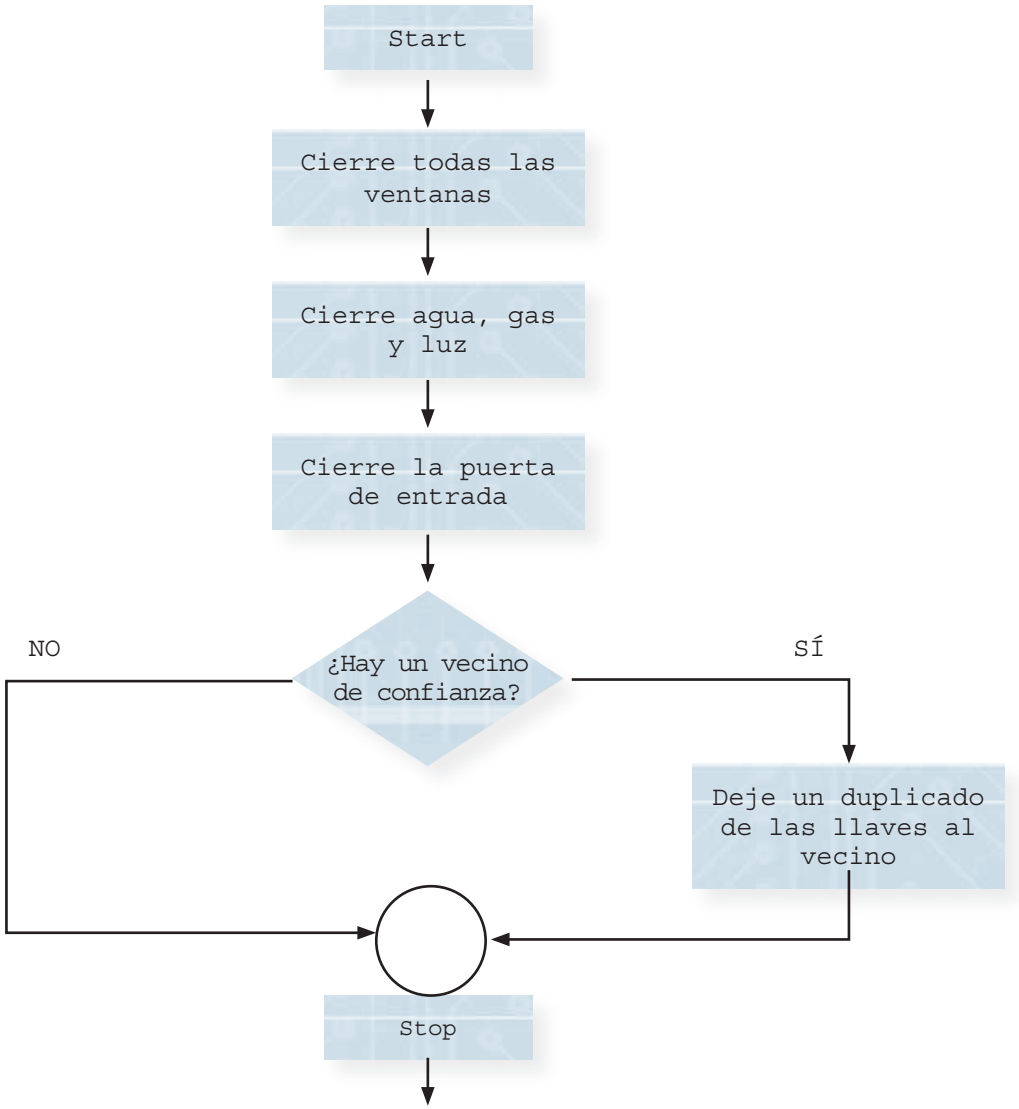


Figura 1.8

En la descripción de los algoritmos o de los programas existen varios formalismos. Pero, de una manera sintética, las reglas comunes a todos para expresar algoritmos, según el paradigma de la programación estructurada, son:

- Un diagrama de flujo se lee de arriba hacia abajo.
- Un diagrama se compone de bloques entre los cuales existen flechas que indican el sentido de lectura o de ejecución.
- Tanto al inicio como al final hay un solo bloque, "`START`" y "`STOP`", respectivamente (véase figura 1.9).

Figura 1.9

- Para las operaciones de entrada o de salida se utilizan los bloques con la forma de un paralelogramo (véase figura 1.10).

Figura 1.10

- Los bloques para hacer asignaciones son rectangulares o cuadrados (véase figura 1.11).

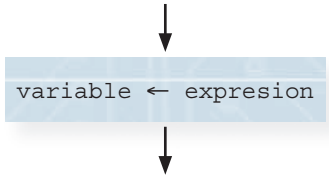


Figura 1.11

- Una decisión tomada con base en una expresión lógica se expresa con un bloque en forma de rombo (véase figura 1.12).

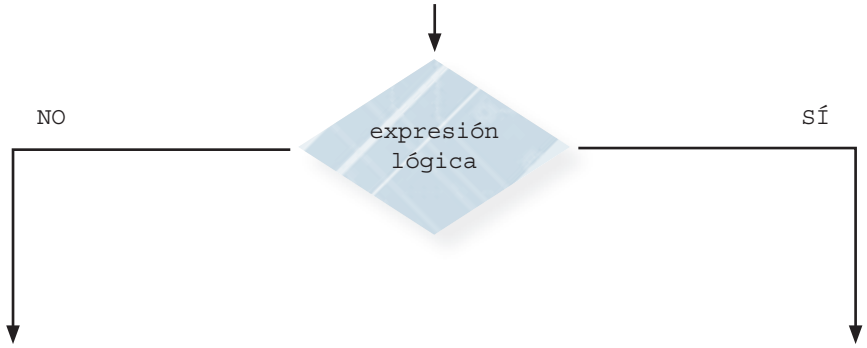


Figura 1.12

La mayoría de las estructuras de la programación estructurada presentadas tienen una transcripción evidente e inmediata en diagramas de flujo:

- La estructura selectiva simple y la estructura selectiva alternativa:

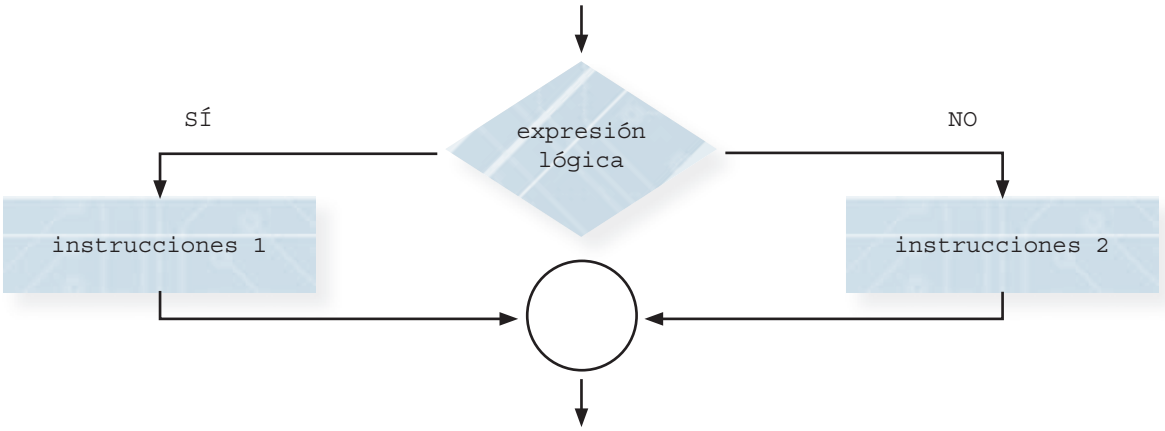


Figura 1.13

Nota: Cualquier rama (SÍ o NO) puede dejarse vacía.

- Estructura iterativa "mientras":

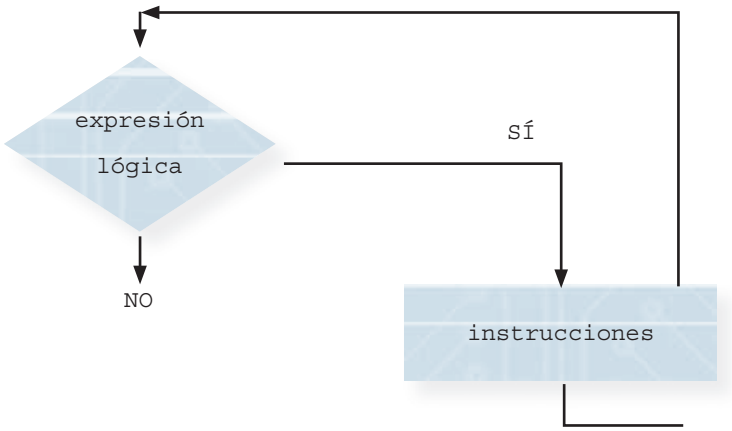


Figura 1.14

- Estructura iterativa "repetir":

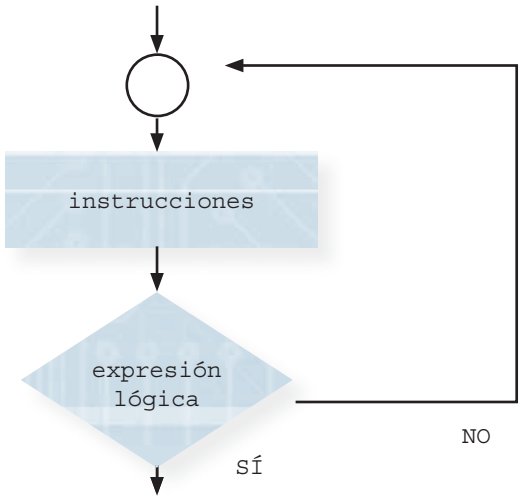


Figura 1.15

En cada una de estas estructuras, "instrucciones" significa cualquier construcción correcta de diagrama de flujo formada para uno solo o más bloques.

Las variables (simples, arreglo o de otro tipo) de un diagrama de flujo pueden considerarse implícitamente declaradas desde sus primeras apariciones, o bien pueden declararse de manera explícita y detallada en un documento anexo al diagrama. De cualquier forma, la primera aparición de una variable debe ser en bloque de entrada o escrita en el lado izquierdo de una asignación. Por ejemplo:

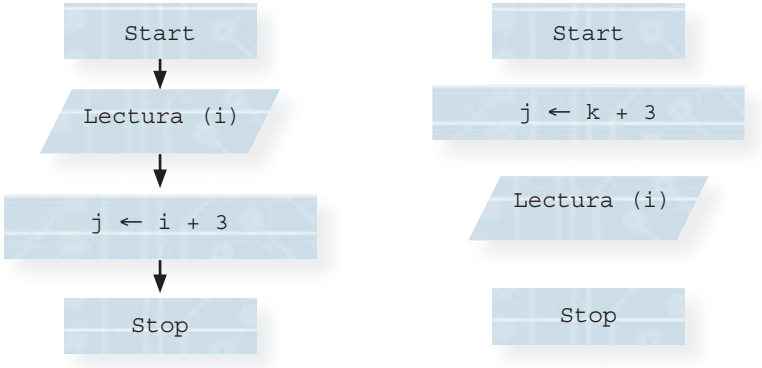


Figura 1.16

El primer diagrama de flujo es correcto, porque en el momento del cálculo de la variable *j*, la variable *i* es conocida y contiene un valor; pero en el segundo diagrama, el uso de la variable *k* del lado derecho de la primera asignación no es correcto, porque aquí la variable *k* aparece por primera vez y no contiene ningún valor.

Los ejemplos de diagramas de flujo que aparecen enseguida, son la resolución de los problemas y ejercicios mostrados como ejemplos a lo largo de todo el capítulo. El diagrama de flujo es, en la mayoría de los casos, una traducción fiel del pseudocódigo. Los diagramas de flujo fueron concebidos y probados con el software Raptor.[8] En el apéndice 2 que se encuentra en el CD-ROM, se presenta este software, la manera cómo hacer los diagramas de flujo y la forma de probarlos.

Ejemplo 1

`Leer una variable x y calcular el cuadrado de x.`

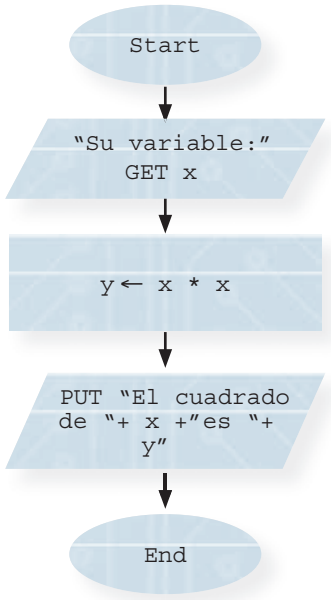


Figura 1.17

[8] Raptor es un software libre disponible en: http://raptor.martincarlisle.com/

Este diagrama de flujo es lineal y muy simple: lectura, cálculo y escritura.

Ejemplo

Verificar si un número entero es o no divisible entre 3.

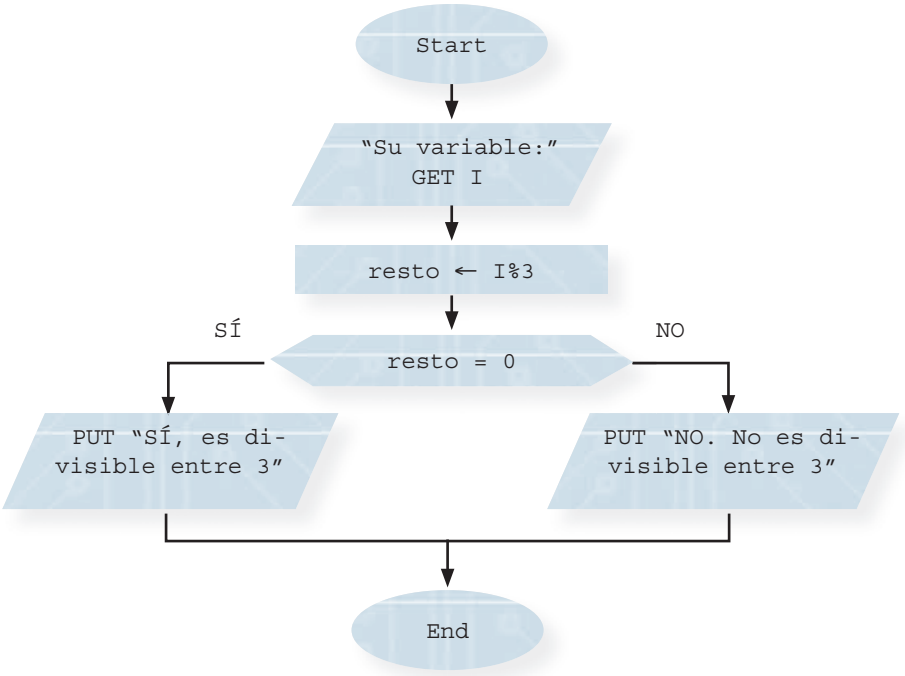


Figura 1.18

Ejemplo 3

Resolver la ecuación de primer grado que tiene su forma matemática más general:

$$ax + b = 0$$

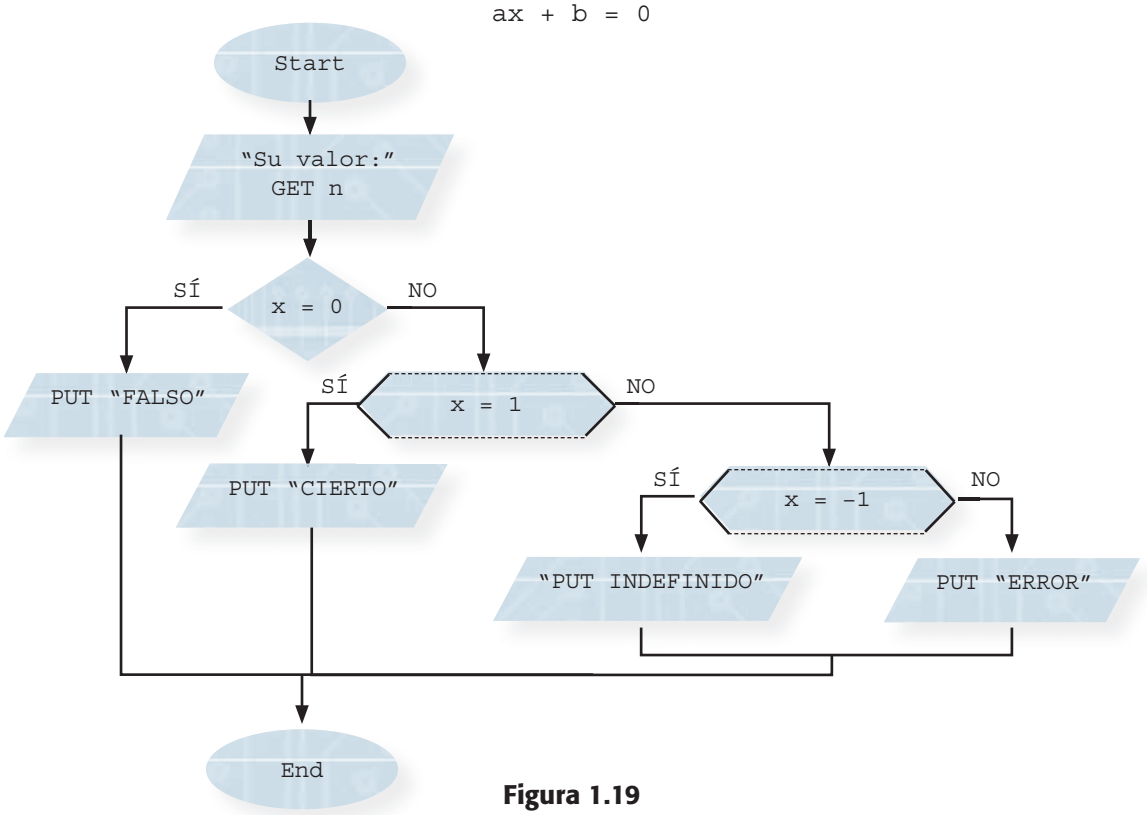


Figura 1.19

Ejemplo 4

Según el valor de una variable de entrada, se escribe:

- *"FALSO" si el valor de la variable es 0.*
- *"CIERTO" si el valor de la variable es 1.*
- *"INDEFINIDO" si el valor de la variable es −1.*
- *"ERROR" en otros casos.*

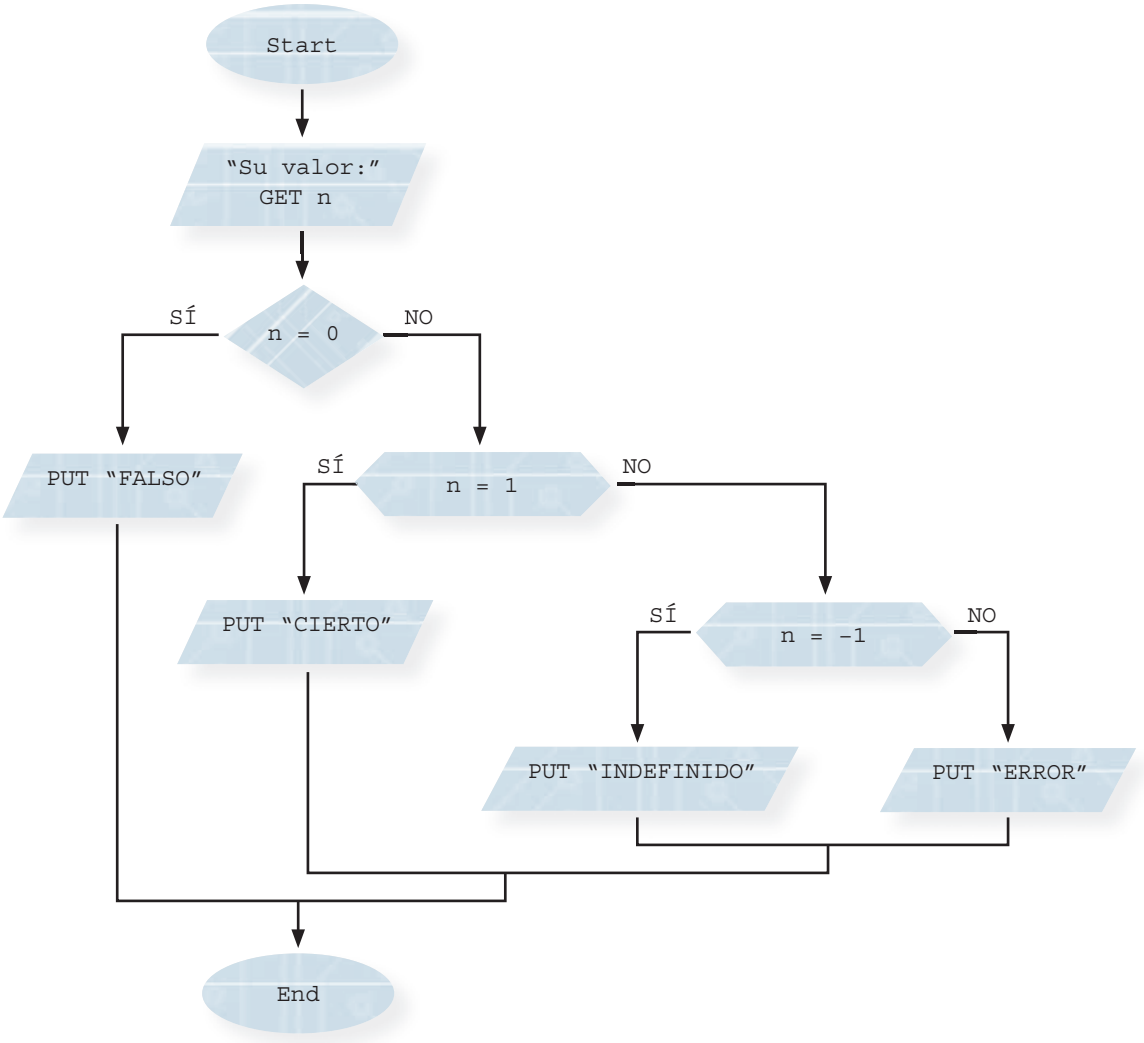


Figura 1.20

Observación: En el software Raptor no se dispone de formalismos para modelar la estructura selectiva "casos"; entonces, se utiliza la estructura alternativa "clásica"; esto es, existe la libertad de poner los bloques de decisión en cualquier orden, aunque es mejor seguir el enunciado del problema.

Ejemplo 5

Buscar el número entero más pequeño, m que es mayor que un número real positivo x, con x ≥ l.

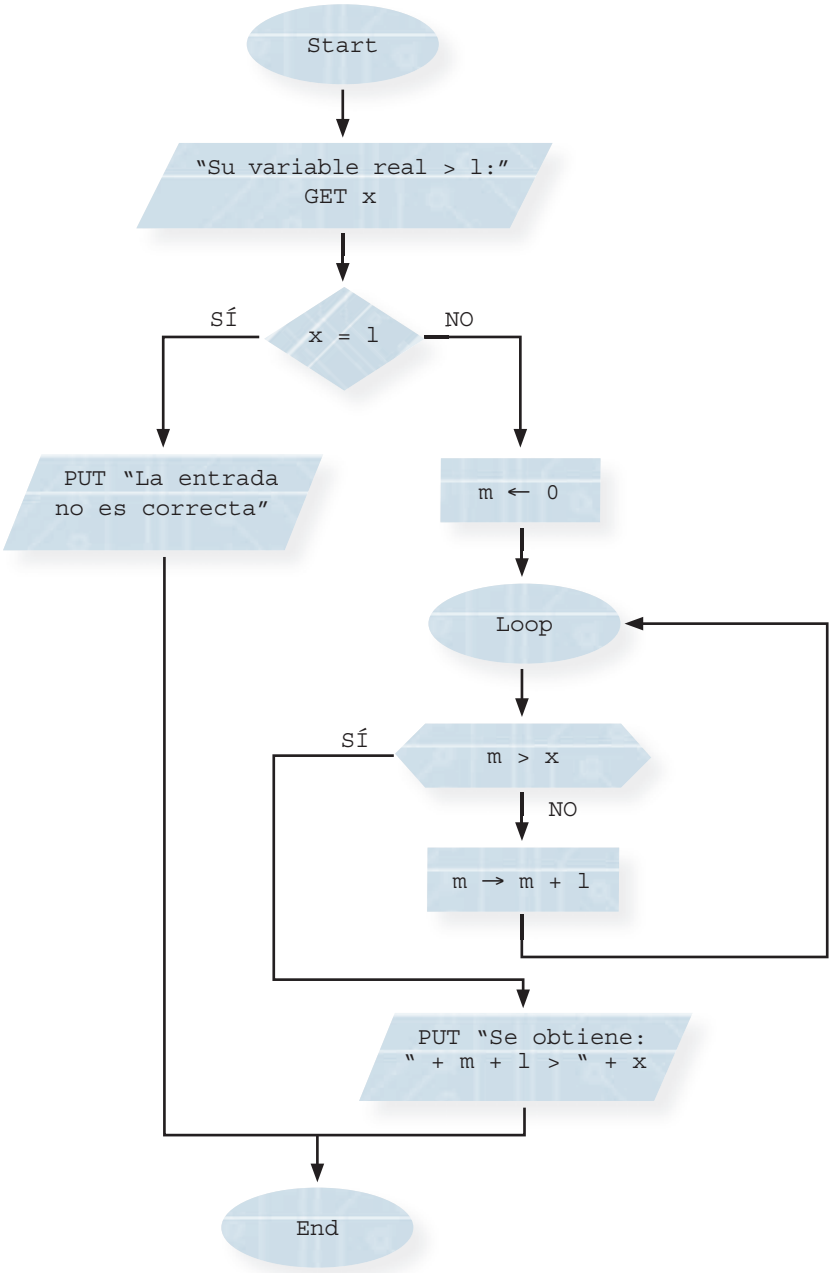


Figura 1.21

Observación: En el diagrama introducimos de manera suplementaria la prueba para certificar si la entrada es correcta; a saber, si $x \geq 1$.

Ejemplo 6

Buscar el número entero más grande de forma 2^k que sea menor que un número real positivo x, con $x \geq 1$.

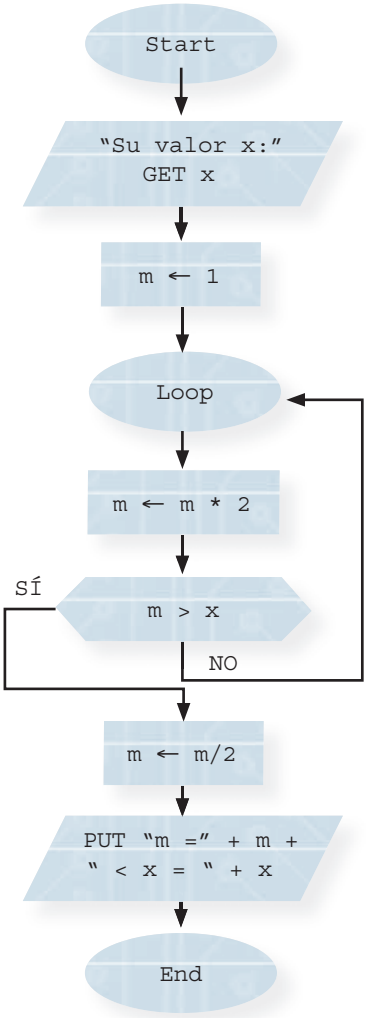


Figura 1.22

Observación: Por razones de tamaño, en este diagrama de flujo hemos renunciando al cálculo de j con $m = 2^j$.

Ejemplo 7

Para un número real x entre 0 y 1 ($0 < x < 1$), una base de numeración b ($b \leq 10$) y un número entero positivo k, buscar las k primeras cifras después del punto decimal de la representación de x en base b.

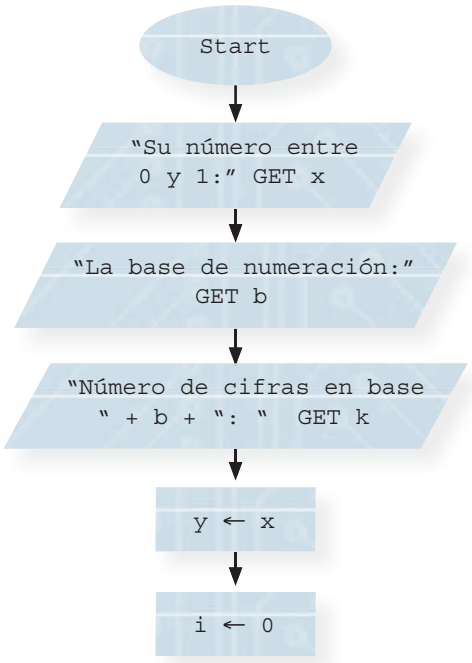


Figura 1.23

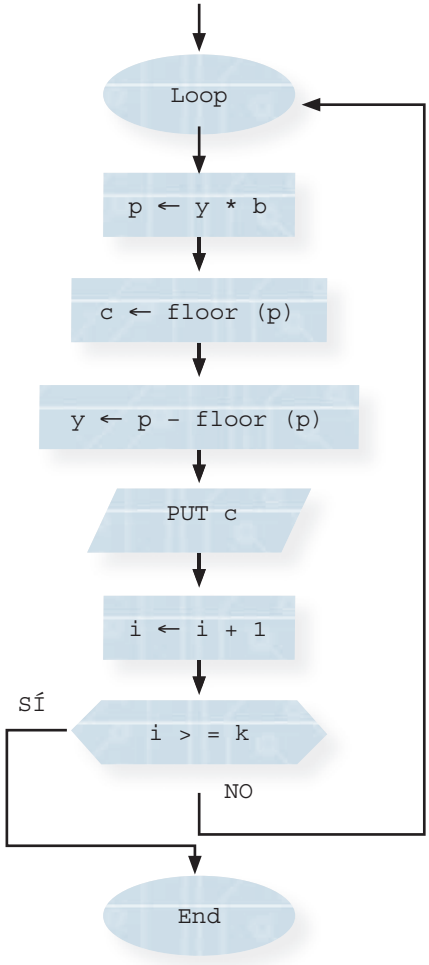


Figura 1.24

Observación: Con respecto al pseudocódigo, cambiamos el orden de obtención de *c* al interior de la estructura iterativa por una razón pedagógica: disponemos de la función `floor()` para el cálculo de la parte entera inferior; entonces, obtenemos la parte fraccionaria por diferencia entre el número y su parte entera. Véase el CD-ROM de apoyo que acompaña este libro, donde se incluye una versión en la que se usa un arreglo para guardar las cifras en base *b* con una escritura más legible del resultado.

Ejemplo 8

Obtener todas las potencias de un número a, desde a^1 hasta a^k, donde a y k son valores de entrada, a es un número real y k es un entero positivo.

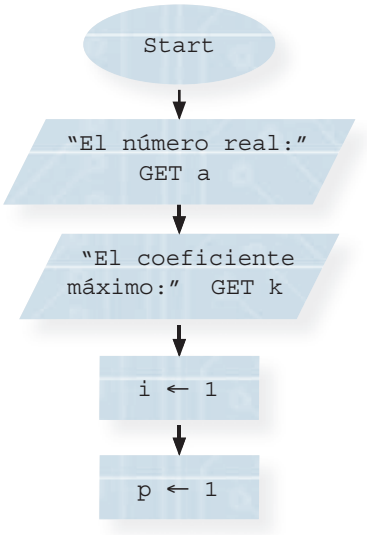


Figura 1.25

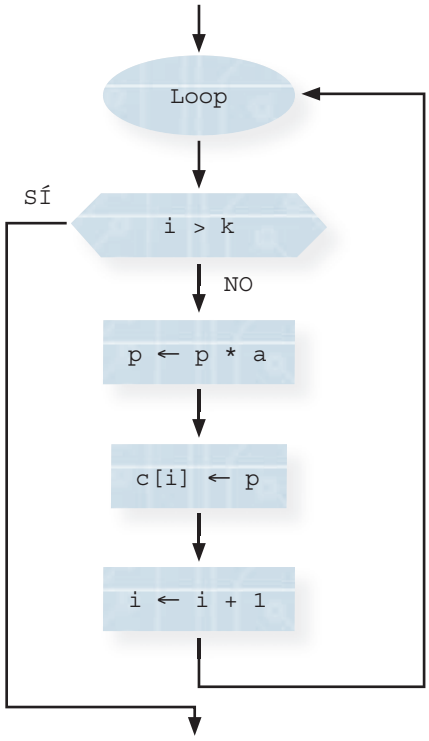


Figura 1.26

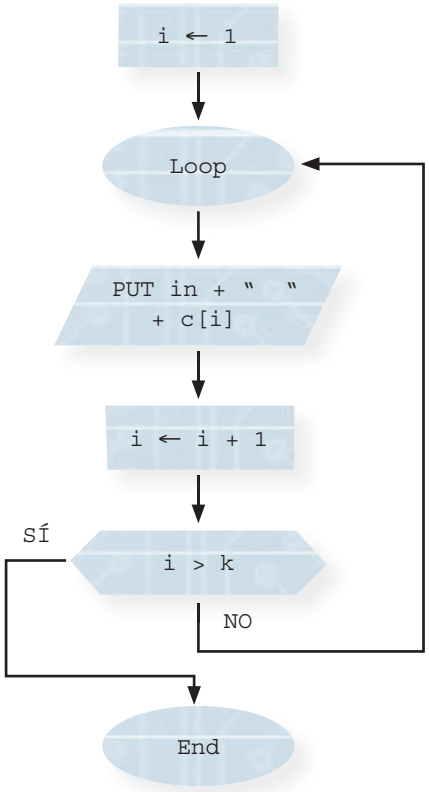


Figura 1.27

Observación: El diagrama de flujo implementa (expresa) la versión con las potencias de *a* almacenadas en un arreglo que se escribe al final, durante otra estructura repetitiva.

Ejemplo 9

Calcular la suma de los elementos de un arreglo que se lee de entrada.

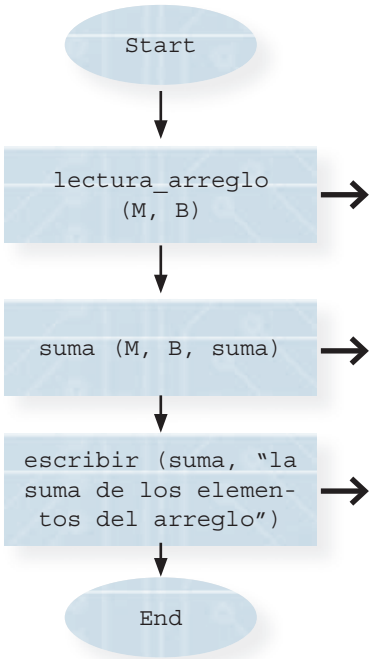


Figura 1.28

Observación: La herramienta que empleamos, únicamente nos permite el uso de procedimientos; entonces, la solución del problema la expresamos como tres llamadas de procedimientos:

1. Por la lectura del arreglo:

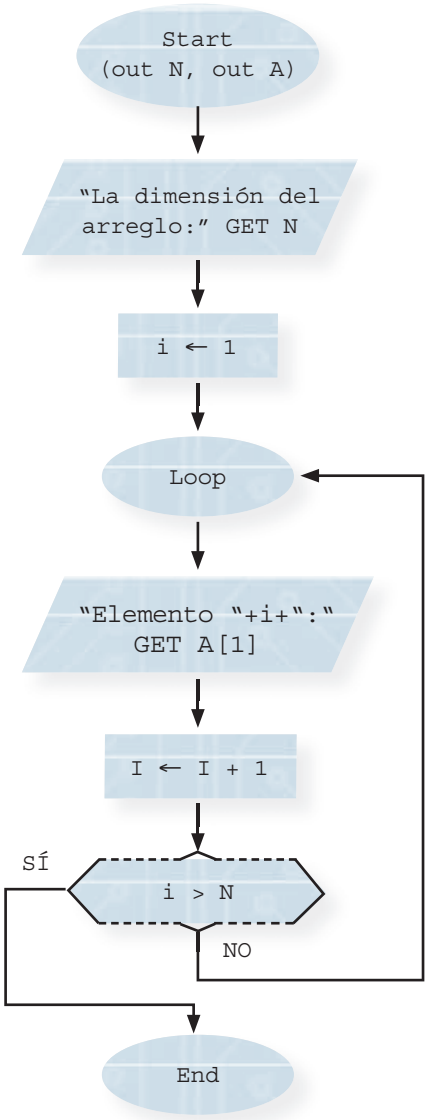


Figura 1.29

2. Por el cálculo de la suma de los elementos del arreglo:

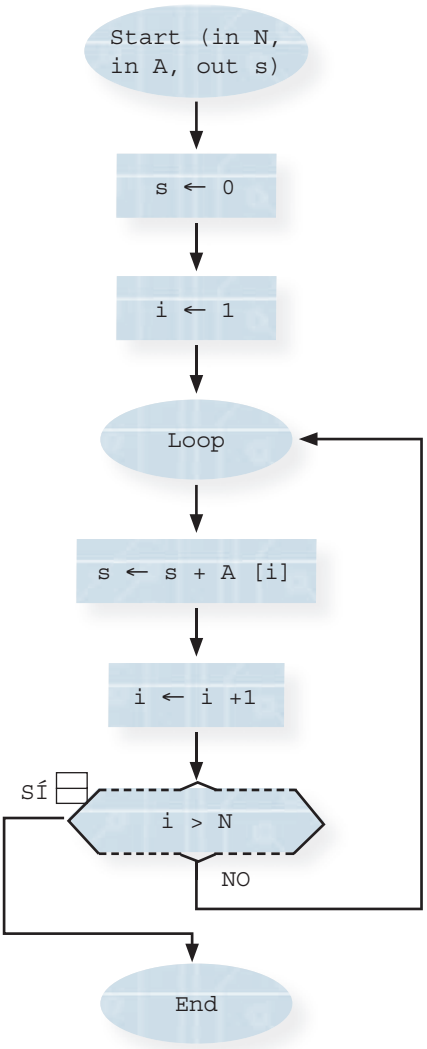


Figura 1.30

3. Por la escritura de un valor y de un mensaje:

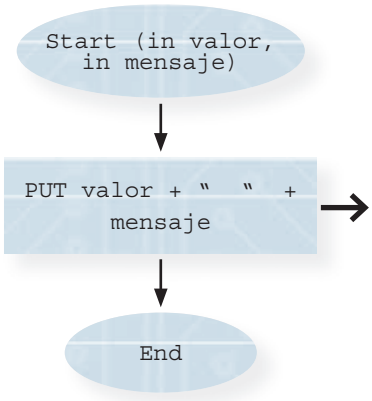


Figura 1.31

Observación: Aquí se puede observar que el nombre del arreglo es *X* y que el nombre de su dimensión es *M*. Estos nombres se usan en las llamadas de los procedimientos de lectura y de cálculo de suma. Así, podemos definir los parámetros de los procedimientos con los nombres que deseamos; en este caso, *N* por el parámetro de dimensión y *A* por el parámetro que guarda el arreglo. Algunos de los parámetros son: de entrada, cuando se calcula la suma de los elementos del arreglo, o de salida, que es el valor calculado de esta suma.

Síntesis del capítulo

La computadora siempre ejecuta órdenes en un formato inteligible para ella; dichas órdenes están agrupadas en un **programa** o **software**. Un programa está escrito en un lenguaje de programación de alto o bajo nivel y traducido en código ejecutable. Por su parte, un software es un conjunto de programas.

El trabajo de realización de un software que resuelve un problema o que responde a una situación está basado en la elaboración de algoritmos. Un **algoritmo** sigue un proceso de elaboración que pasa por las siguientes fases:

1. **Definición.** Se especifica el propósito del algoritmo.
2. **Análisis.** Se analizan el problema y sus características; se determinan las entradas y las salidas del problema, y se elige la solución más conveniente, si hay varias, o se propone una nueva.
3. **Diseño.** Se plasma la solución del problema; aquí se emplea una herramienta de diseño: el diagrama de flujo y el pseudocódigo.
4. **Implementación.** Se realiza el programa y se hacen varias pruebas; el programa se edita con editores de texto y se compila o se interpreta a fin de crear el ejecutable o ejecutar el código.

Los programas se escriben en un lenguaje de programación. Hay varios paradigmas de programación y una multitud de lenguajes de programación que tienen uno o varios paradigmas. La elección del lenguaje de programación depende principalmente del tipo de problema a resolver, de la computadora y de otros dispositivos físicos que se utilizarán.

Los programas contienen variables. Una **variable** tiene un tipo y un nombre que debe ser único. Según los paradigmas el lenguaje, la asignación de una zona de memoria para la variable se hace en la memoria (memoria central, en la mayoría de los lenguaje) de manera estática o dinámica.

El pseudocódigo y los diagramas de flujo son herramientas de diseño de algoritmos más o menos equivalentes, que tienen el paradigma de programación imperativa y estructurada.

Las estructuras de paradigma que se conocen son:

- Estructura secuencial.
- Estructura alternativa.
- Estructura iterativa.
- Funciones y procedimientos.

Las variables que se usan en el pseudocódigo o en el diagrama de flujo pueden ser simples, de tipo arreglo o de otro tipo, descrito por el programador. Cada variable posee un nombre único y no ambiguo y tiene reservada una zona de memoria en la cual se almacena el valor de la variable.

Bibliografía

- Knuth, Donald, *El arte de programar ordenadores*, Vol. I, "Algoritmos fundamentales", Editorial Reverté, Barcelona, Bogotá, México, 1986.
- Cedano Olvera, Marco Alfredo y otros, *Fundamentos de computación para ingenieros*, Grupo Editorial Patria, México, 2010.
- Alfred V., Sethi, Ravi y Ullman, Jeffrey D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Estados Unidos, 1986.

Ejercicios y problemas

1. ¿Cuál es la diferencia entre un programa y un algoritmo?

2. ¿Es posible escribir directamente un programa para la resolución de un problema? ¿Es útil?

3. Construir el árbol de evaluación y luego escribir la expresión en lenguaje informático en forma de infijo de:

a) $ab + bd$

b) $a^2 - 10^3$

c) $\dfrac{a - \sqrt{b^2 - 4ac}}{2b}$

d) $a + b < c$ y $2c < 4a + 3b$

4. Escribir la expresión lógica que corresponde a la expresión $a \neq 0$ y $b^2 - 4ac > 0$ matemática siguiente:

5. Proponer nombres de variables para resolver una ecuación de segundo grado.

6. Si los coeficientes algebraicos de una ecuación de segundo grado son números reales, proponer tipos para las variables elegidas en el problema anterior.

7. Si los coeficientes de la ecuación $x^2 - by^2 = 0$ son números enteros, proponer nombres y tipos de variables para buscar soluciones enteras.

8. ¿Por qué la definición de una función parece tan diferente entre los lenguajes C, PASCAL y PL/SQL, por un lado, y PROLOG o LISP, por el otro? ¿Por qué los primeros ejemplos son tan parecidos (lenguajes C, PASCAL y PL/SQL)?