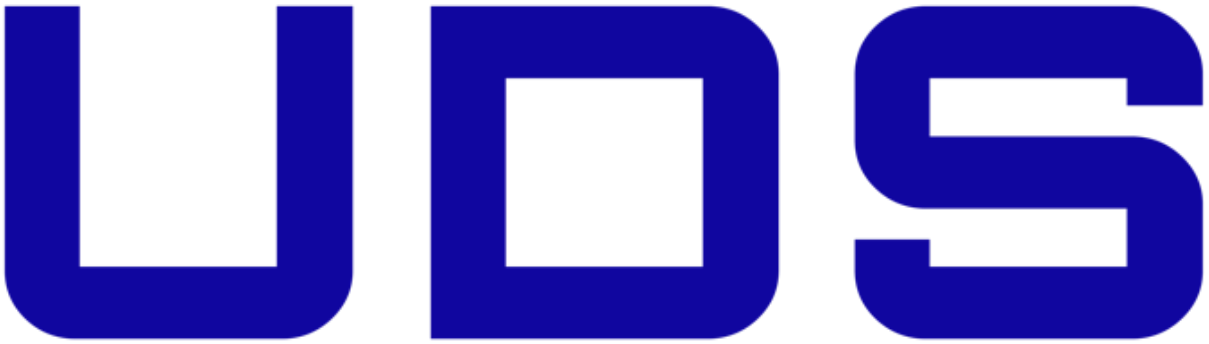


LIBRO

FUNDAMENTOS Y LÓGICA DE PROGRAMACIÓN

INGENIERÍA EN SISTEMAS COMPUTACIONALES

3°CUATRIMESTRE

Marco Estratégico de Referencia

ANTECEDENTES HISTORICOS

Nuestra Universidad tiene sus antecedentes de formación en el año de 1979 con el inicio de actividades de la normal de educadoras "Edgar Robledo Santiago", que en su momento marcó un nuevo rumbo para la educación de Comitán y del estado de Chiapas. Nuestra escuela fue fundada por el Profesor de Primaria Manuel Albores Salazar con la idea de traer Educación a Comitán, ya que esto representaba una forma de apoyar a muchas familias de la región para que siguieran estudiando.

En el año 1984 inicia actividades el CBTiS Moctezuma Ilhuicamina, que fue el primer bachillerato tecnológico particular del estado de Chiapas, manteniendo con esto la visión en grande de traer Educación a nuestro municipio, esta institución fue creada para que la gente que trabajaba por la mañana tuviera la opción de estudiar por las tarde.

La Maestra Martha Ruth Alcázar Mellanes es la madre de los tres integrantes de la familia Albores Alcázar que se fueron integrando poco a poco a la escuela formada por su padre, el Profesor Manuel Albores Salazar; Víctor Manuel Albores Alcázar en septiembre de 1996 como chofer de transporte escolar, Karla Fabiola Albores Alcázar se integró como Profesora en 1998, Martha Patricia Albores Alcázar en el departamento de finanzas en 1999.

En el año 2002, Víctor Manuel Albores Alcázar formó el Grupo Educativo Albores Alcázar S.C. para darle un nuevo rumbo y sentido empresarial al negocio familiar y en el año 2004 funda la Universidad Del Sureste.

La formación de nuestra Universidad se da principalmente porque en Comitán y en toda la región no existía una verdadera oferta Educativa, por lo que se veía urgente la creación de una institución de Educación superior, pero que estuviera a la altura de las exigencias de los jóvenes que tenían intención de seguir estudiando o de los profesionistas para seguir preparándose a través de estudios de posgrado.

Nuestra Universidad inició sus actividades el 18 de agosto del 2004 en las instalaciones de la 4ª avenida oriente sur no. 24, con la licenciatura en Puericultura, contando con dos grupos de cuarenta

alumnos cada uno. En el año 2005 nos trasladamos a nuestras propias instalaciones en la carretera Comitán – Tzimol km. 57 donde actualmente se encuentra el campus Comitán y el Corporativo UDS, este último, es el encargado de estandarizar y controlar todos los procesos operativos y Educativos de los diferentes Campus, Sedes y Centros de Enlace Educativo, así como de crear los diferentes planes estratégicos de expansión de la marca a nivel nacional e internacional.

Nuestra Universidad inició sus actividades el 18 de agosto del 2004 en las instalaciones de la 4ª avenida oriente sur no. 24, con la licenciatura en Puericultura, contando con dos grupos de cuarenta alumnos cada uno. En el año 2005 nos trasladamos a nuestras propias instalaciones en la carretera Comitán – Tzimol km. 57 donde actualmente se encuentra el campus Comitán y el corporativo UDS, este último, es el encargado de estandarizar y controlar todos los procesos operativos y educativos de los diferentes campus, así como de crear los diferentes planes estratégicos de expansión de la marca.

MISIÓN

Satisfacer la necesidad de Educación que promueva el espíritu emprendedor, aplicando altos estándares de calidad Académica, que propicien el desarrollo de nuestros alumnos, Profesores, colaboradores y la sociedad, a través de la incorporación de tecnologías en el proceso de enseñanza-aprendizaje.

VISIÓN

Ser la mejor oferta académica en cada región de influencia, y a través de nuestra Plataforma Virtual tener una cobertura Global, con un crecimiento sostenible y las ofertas académicas innovadoras con pertinencia para la sociedad.

VALORES

- Disciplina
- Honestidad
- Equidad
- Libertad

ESCUDO

El escudo de la UDS, está constituido por tres líneas curvas que nacen de izquierda a derecha formando los escalones al éxito. En la parte superior está situado un cuadro motivo de la abstracción de la forma de un libro abierto.

ESLOGAN

"Mi Universidad"

ALBORES

Es nuestra mascota, un Jaguar. Su piel es negra y se distingue por ser líder, trabaja en equipo y obtiene lo que desea. El ímpetu, extremo valor y fortaleza son los rasgos que distinguen.

Fundamentos y lógica de programación

Objetivo de la materia:

Analizar y solucionar problemas informáticos y representar su solución mediante herramientas de software de programación.

UNIDAD I INTRODUCCIÓN A LA PROGRAMACIÓN

1.1 Lenguajes de programación.
1.1.1 Traductores de lenguaje: proceso de traducción de un programa.
1.1.2 La compilación y sus fases.
1.2 Evolución de los lenguajes de programación.
1.3 Paradigmas de programación.
1.4 Historia de los lenguajes de programación.
1.5 Fases en la resolución de problemas.
1.6 Análisis del problema.
1.7 Diseño de algoritmo.
1.8 Herramientas de programación.
1.9 Codificación de un programa.
1.10 Compilación y ejecución de un programa.
1.11 Verificación y depuración de un programa.
1.12 Documentación y mantenimiento.

UNIDAD II TIPOS DE PROGRAMACIÓN Y ALGORITMOS

2.1 Programación modular.
2.2 Programación estructurada.
2.3 Programación orientada a objetos.
2.4 Abstracción.
2.5 Encapsulación y ocultación de datos.
2.6 Objetos.
2.7 Clases.
2.8 Generalización y especialización: herencia.
2.9 Reusabilidad.
2.10 Polimorfismo.
2.11 Algoritmos.
2.11.1 Concepto y características de los algoritmos.
2.11.2 Diseño del algoritmo.
2.11.3 Escritura de algoritmos.
2.11.4 Representación gráfica de los algoritmos.
2.11.5 Pseudocódigo.
2.11.6 Diagrama de flujo.

UNIDAD III ESTRUCTURA GENERAL DE UN PROGRAMA

3.1 Concepto de programa.
3.2 Partes constitutivas de un programa.
3.3 Instrucciones y tipos de instrucciones.
3.4 Elementos básicos de un programa.
3.5 Datos, tipos de datos y operaciones primitivas.
3.5.1 Datos numéricos.
3.5.2 Datos lógicos (booleanos).
3.5.3 Datos tipo carácter y tipo cadena.
3.6 Constantes y variables.
3.7 Expresiones.
3.7.1 Expresiones aritméticas.
3.7.2 Reglas de prioridad.
3.7.3 Expresiones lógicas (booleanas)
3.8 La operación de asignación.
3.9 Entrada y salida de información.
3.10 Escritura de algoritmos/programas.
3.11 Cabecera del programa.
3.12 Declaración de variables.
3.13 Declaración de constantes numéricas.
3.14 Declaración de constantes y variables carácter.
3.15 Comentarios.

UNIDAD IV ESTRUCTURAS DE CONTROL DE UN PROGRAMA

4.1 Estructura secuencial.
4.2 Estructuras selectivas.
4.3 Alternativa simple (si-entonces/if-then).
4.4 Alternativa doble (si-entonces-sino/if-then-else).
4.5 Alternativa múltiple (según_sea, caso de/case).
4.6 Estructuras de decisión anidadas (en escalera).
4.7 Estructura mientras (“while”).
4.8 Estructura hacer-mientras ("do-while").
4.9 Estructura repetir ("repeat").
4.10 Estructura desde/para ("for").
4.11 Sentencias de salto interrumpir (break) y continuar (continue).

Índice

UNIDAD I INTRODUCCIÓN A LA PROGRAMACIÓN

1.1 Lenguajes de programación.

Como se ha visto en el apartado anterior, para que un procesador realice un proceso se le debe suministrar en primer lugar un algoritmo adecuado. El procesador debe ser capaz de interpretar el algoritmo, lo que significa:

• comprender las instrucciones de cada paso,

• realizar las operaciones correspondientes.

Cuando el procesador es una computadora, el algoritmo se ha de expresar en un formato que se denomina programa, ya que el pseudocódigo o el diagrama de flujo no son comprensibles por la computadora, aunque pueda entenderlos cualquier programador. Un programa se escribe en un lenguaje de programación y las operaciones que conducen a expresar un algoritmo en forma de programa se llaman programación. Así pues, los lenguajes utilizados para escribir programas de computadoras son los lenguajes de programación y programadores son los escritores y diseñadores de programas. El proceso de traducir un algoritmo en pseudocódigo a un lenguaje de programación se denomina codificación, y el algoritmo escrito en un lenguaje de programación se denomina código fuente.

En la realidad la computadora no entiende directamente los lenguajes de programación, sino que se requiere un programa que traduzca el código fuente a otro lenguaje que sí entiende la máquina directamente, pero muy complejo para las personas; este lenguaje se conoce como lenguaje máquina y el código correspondiente código máquina.

Los programas que traducen el código fuente escrito en un lenguaje de programación —tal como C++— a código máquina se denominan traductores. El proceso de conversión de un algoritmo escrito en pseudocódigo hasta un programa ejecutable comprensible por la máquina.

Hoy en día, la mayoría de los programadores emplean lenguajes de programación como C++, C, C#, Java, Visual Basic, XML, HTML, Perl, PHP, JavaScript..., aunque todavía se

utilizan, sobre todo profesionalmente, los clásicos COBOL, FORTRAN, Pascal o el mítico BASIC. Estos lenguajes se denominan lenguajes de alto nivel y permiten a los profesionales resolver problemas convirtiendo sus algoritmos en programas escritos en alguno de estos lenguajes de programación.

Los lenguajes de programación se utilizan para escribir programas. Los programas de las computadoras modernas constan de secuencias de instrucciones que se codifican como secuencias de dígitos numéricos que podrán entender dichas computadoras. El sistema de codificación se conoce como lenguaje máquina que es el lenguaje nativo de una computadora. Desgraciadamente la escritura de programas en lenguaje máquina es una tarea tediosa y difícil ya que sus instrucciones son secuencias de 0 y 1 (patrones de bit, tales como 11110000, 01110011...) que son muy difíciles de recordar y manipular por las personas. En consecuencia, se necesitan lenguajes de programación "amigables con el programador" que permitan escribir los programas para poder "charlar" con facilidad con las computadoras. Sin embargo, las computadoras sólo entienden las instrucciones en lenguaje máquina, por lo que será preciso traducir los programas resultantes a lenguajes de máquina antes de que puedan ser ejecutadas por ellas.

Cada lenguaje de programación tiene un conjunto o "juego" de instrucciones (acciones u operaciones que debe realizar la máquina) que la computadora podrá entender directamente en su código máquina o bien se traducirán a dicho código máquina. Las instrucciones básicas y comunes en casi todos los lenguajes de programación son:

• Instrucciones de entrada/salida. Instrucciones de transferencia de información entre dispositivos periféricos y la memoria central, tales como "leer de..." o bien "escribir en...".

• Instrucciones de cálculo. Instrucciones para que la computadora pueda realizar operaciones aritméticas.

• Instrucciones de control. Instrucciones que modifican la secuencia de la ejecución del programa.

Además de estas instrucciones y dependiendo del procesador y del lenguaje de programación existirán otras que conformarán el conjunto de instrucciones y junto con las reglas de sintaxis permitirán escribir los programas de las computadoras. Los principales tipos de lenguajes de programación son:

• Lenguajes máquina.

• Lenguajes de bajo nivel (ensambladores).

• Lenguajes de alto nivel.

Traductores de lenguaje: proceso de traducción de un programa.

El proceso de traducción de un programa fuente escrito en un lenguaje de alto nivel a un lenguaje máquina comprensible por la computadora, se realiza mediante programas llamados "traductores". Los traductores de lenguaje son programas que traducen a su vez los programas fuente escritos en lenguajes de alto nivel a código máquina. Los traductores se dividen en compiladores e intérpretes.

Intérpretes

Un intérprete es un traductor que toma un programa fuente, lo traduce y, a continuación, lo ejecuta. Los programas intérpretes clásicos como BASIC, prácticamente ya no se utilizan, más que en circunstancias especiales. Sin embargo, está muy extendida la versión interpretada del lenguaje Smalltalk, un lenguaje orientado a objetos puro. El sistema de traducción consiste en: traducir la primera sentencia del programa a lenguaje máquina, se detiene la traducción, se ejecuta la sentencia; a continuación, se traduce la siguiente sentencia, se detiene la traducción, se ejecuta la sentencia y así sucesivamente hasta terminar el programa.

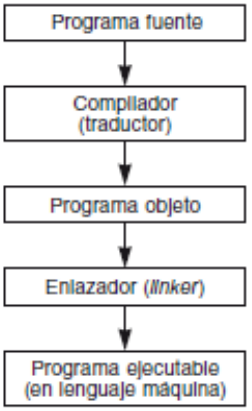
Compiladores

Un compilador es un programa que traduce los programas fuente escritos en lenguaje de alto nivel a lenguaje máquina. La traducción del programa completo se realiza en una sola operación denominada compilación del programa; es decir, se traducen todas las instrucciones del programa en un solo bloque. El programa compilado y depurado (eliminados los errores del código fuente) se denomina programa ejecutable porque ya se puede ejecutar directamente y cuantas veces se desee; sólo deberá volver a compilarse de nuevo en el caso de que se modifique alguna instrucción del programa. De este modo el

programa ejecutable no necesita del compilador para su ejecución. Los traductores de lenguajes típicos más utilizados son: C, C++, Java, C#, Pascal, FORTRAN y COBOL.

1.1.2 La compilación y sus fases.

La compilación es el proceso de traducción de programas fuente a programas objeto. El programa objeto obtenido de la compilación ha sido traducido normalmente a código máquina.



Para conseguir el programa máquina real se debe utilizar un programa llamado montador o enlazador (linker). El proceso de montaje conduce a un programa en lenguaje máquina directamente ejecutable El proceso de ejecución de un programa escrito en un lenguaje de programación y mediante un compilador suele tener los siguientes pasos:

1. Escritura del programa fuente con un editor (programa que permite a una computadora actuar de modo similar a una máquina de escribir electrónica) y guardarlo en un dispositivo de almacenamiento (por ejemplo, un disco).

2. Introducir el programa fuente en memoria.

3. Compilar el programa con el compilador seleccionado.

4. Verificar y corregir errores de compilación (listado de errores).

5. Obtención del programa objeto.

6. El enlazador (linker) obtiene el programa ejecutable.

7. Se ejecuta el programa y, si no existen errores, se tendrá la salida del programa.

1.2 Evolución de los lenguajes de programación.

En la década de los cuarenta cuando nacían las primeras computadoras digitales el lenguaje que se utilizaba para programar era el lenguaje máquina que traducía directamente el código

máquina (código binario) comprensible para las computadoras. Las instrucciones en lenguaje máquina dependían de cada computadora y debido a la dificultad de su escritura, los investigadores de la época simplificaron el proceso de programación desarrollando sistemas de notación en los cuales las instrucciones se representaban en formatos nemónicos (nemotécnicos) en vez de en formatos numéricos que eran más difíciles de recordar. Por ejemplo, mientras la instrucción

```
Mover el contenido del registro 4 al registro 8
```

se podía expresar en lenguaje máquina como

```
4048 o bien 0010 0000 0010 1000
```

en código nemotécnico podía aparecer como

```
MOV R5, R6
```

Para convertir los programas escritos en código nemotécnico a lenguaje máquina, se desarrollaron programas ensambladores (assemblers). Es decir, los ensambladores son programas que traducen otros programas escritos en código nemotécnico en instrucciones numéricas en lenguaje máquina que son compatibles y legibles por la máquina. Estos programas de traducción se llaman ensambladores porque su tarea es ensamblar las instrucciones reales de la máquina con los nemotécnicos e identificadores que representan las instrucciones escritas en ensamblador. A estos lenguajes se les denominó de segunda generación, reservando el nombre de primera generación para los lenguajes de máquina.

En la década de los cincuenta y sesenta comenzaron a desarrollarse lenguajes de programación de tercera generación que diferían de las generaciones anteriores en que sus instrucciones o primitivas eran de alto nivel (comprensibles por el programador, como si fueran lenguajes naturales) e independientes de la máquina. Estos lenguajes se llamaron lenguajes de alto nivel. Los ejemplos más conocidos son FORTRAN (FORmula TRANslator) que fue desarrollado para aplicaciones científicas y de ingeniería, y COBOL (COmmon Business-Oriented Language), que fue desarrollado por la U.S. Navy de Estados Unidos, para aplicaciones de gestión o administración. Con el paso de los años aparecieron nuevos lenguajes tales como Pascal, BASIC, C, C++, Ada, Java, C#, HTML, XML...

Los lenguajes de programación de alto nivel se componen de un conjunto de instrucciones o primitivas más fáciles de escribir y recordar su función que los lenguajes máquina y ensamblador. Sin embargo, los programas escritos en un lenguaje de alto nivel, como C o Java necesitan ser traducidos a código máquina; para ello se requiere un programa denominado traductor. Estos programas de traducción se denominaron técnicamente, compiladores. De este modo existen compiladores de C, FORTRAN, Pascal, Java, etc.

También surgió una alternativa a los traductores compiladores como medio de implementación de lenguajes de tercera generación que se denominaron intérpretes. Estos programas eran similares a los traductores excepto que ellos ejecutaban las instrucciones a medida que se traducían, en lugar de guardar la versión completa traducida para su uso posterior. Es decir, en vez de producir una copia de un programa en lenguaje máquina que se ejecuta más tarde (este es el caso de la mayoría de los lenguajes, C, C++, Pascal, Java...), un intérprete ejecuta realmente un programa desde su formato de alto nivel, instrucción a instrucción. Cada tipo de traductor tiene sus ventajas e inconvenientes, aunque hoy día prácticamente los traductores utilizados son casi todos compiladores por su mayor eficiencia y rendimiento.

Sin embargo, en el aprendizaje de programación se suele comenzar también con el uso de los lenguajes algorítmicos, similares a los lenguajes naturales, mediante instrucciones escritas en pseudocódigo (o seudocógido) que son palabras o abreviaturas de palabras escritas en inglés, español, portugués, etc. Posteriormente se realiza la conversión al lenguaje de alto nivel que se vaya a utilizar realmente en la computadora, tal como C, C++ o Java. Esta técnica facilita la escritura de algoritmos como paso previo a la programación.

1.3 Paradigmas de programación.

La evolución de los lenguajes de programación ha ido paralela a la idea de paradigma de programación: enfoques alternativos a los procesos de programación. En realidad un paradigma de programación representa fundamentalmente enfoques diferentes para la construcción de soluciones a problemas y por consiguiente afectan al proceso completo de desarrollo de software. Los paradigmas de programación clásicos son: procedimental (o imperativo), funcional, declarativo y orientado a objetos. En la Figura 1.21 se muestra la

evolución de los paradigmas de programación y los lenguajes asociados a cada paradigma [BROOKSHEAR 04]24.

Lenguajes imperativos (procedimentales)

El paradigma imperativo o procedimental representa el enfoque o método tradicional de programación. Un lenguaje imperativo es un conjunto de instrucciones que se ejecutan una por una, de principio a fin, de modo secuencial excepto cuando intervienen instrucciones de salto de secuencia o control. Este paradigma define el proceso de programación como el desarrollo de una secuencia de órdenes (comandos) que manipulan los datos para producir los resultados deseados. Por consiguiente, el paradigma imperativo señala un enfoque del proceso de programación mediante la realización de un algoritmo que resuelve de modo manual el problema y a continuación expresa ese algoritmo como una secuencia de órdenes. En un lenguaje procedimental cada instrucción es una orden u órdenes para que la computadora realice alguna tarea específica. Los lenguajes de programación procedimentales, por excelencia, son FORTRAN, COBOL, Pascal, BASIC, ALGOL, C y Ada (aunque sus últimas versiones ya tienen un carácter completamente orientado a objetos).

Lenguajes declarativos

En contraste con el paradigma imperativo el paradigma declarativo solicita al programador que describa el problema en lugar de encontrar una solución algorítmica al problema; es decir, un lenguaje declarativo utiliza el principio del razonamiento lógico para responder a las preguntas o cuestiones consultadas. Se basa en la lógica formal y en el cálculo de predicados de primer orden. El razonamiento lógico se basa en la deducción. El lenguaje declarativo por excelencia es Prolog.

Lenguajes orientados a objetos

El paradigma orientado a objetos se asocia con el proceso de programación llamado programación orientada a objetos (POO)25 consistente en un enfoque totalmente distinto al proceso procedimental. El enfoque orientado a objetos guarda analogía con la vida real. El desarrollo de software OO se basa en el diseño y construcción de objetos que se componen a su vez de datos y operaciones que manipulan esos datos. El programador define en primer lugar los objetos del problema y a continuación los datos y operaciones que actuarán sobre esos datos. Las ventajas de la programación orientada a objetos se

derivan esencialmente de la estructura modular existente en la vida real y el modo de respuesta de estos módulos u objetos a mensajes o eventos que se producen en cualquier instante.

Los orígenes de la POO se remontan a los Tipos Abstractos de Datos como parte constitutiva de una estructura de datos. En este libro se dedicará un capítulo completo al estudio del TAD como origen del concepto de programación denominado objeto.

C++ lenguaje orientado a objetos, por excelencia, es una extensión del lenguaje C y contiene las tres propiedades más importantes: encapsulamiento, herencia y polimorfismo. Smalltalk es otro lenguaje orientado a objetos muy potente y de gran impacto en el desarrollo del software orientado a objetos que se ha realizado en las últimas décadas.

Hoy día Java y C# son herederos directos de C++ y C, y constituyen los lenguajes orientados a objetos más utilizados en la industria del software del siglo XXI. Visual Basic y VB.Net son otros lenguajes orientados a objetos, no tan potentes como los anteriores pero extremadamente sencillos y fáciles de aprender.

1.4 Historia de los lenguajes de programación.

La historia de la computación ha estado asociada indisolublemente a la aparición y a la historia de lenguajes de programación de computadoras26. La Biblia de los lenguajes ha sido una constante en el desarrollo de la industria del software y en los avances científicos y tecnológicos. Desde el año 1642 en que Blaise Pascal, inventó La Pascalina, una máquina que ayudaba a contar mediante unos dispositivos de ruedas, se han sucedido numerosos inventos que han ido evolucionando, a medida que se programaban mediante códigos de máquina, lenguajes ensambladores, hasta llegar a los lenguajes de programación de alto nivel en los que ya no se dependía del hardware de la máquina sino de la capacidad de abstracción del programador y de la sintaxis, semántica y potencia del lenguaje.

En la década de los cincuenta, IBM diseñó el primer lenguaje de programación comercial de alto nivel y concebido para resolver problemas científicos y de ingeniería (FORTRAN, 1954). Todavía hoy, muchos científicos e ingenieros siguen utilizando FORTRAN en sus versiones más recientes FORTRAN 77 y FORTRAN 90. En 1959, la doctora y almirante,

Grace Hopper, lideró el equipo que desarrolló COBOL, el lenguaje por excelencia del mundo de la gestión y de los negocios hasta hace muy poco tiempo; aunque todavía el mercado sigue demandando programadores de COBOL ya que numerosas aplicaciones comerciales siguen corriendo en este lenguaje.

Una enumeración rápida de lenguajes de programación que han sido o son populares y los años en que aparecieron es la siguiente:

Década 50	Década 60	Década 70	Década 80	Década 90	Década 00
FORTRAN (1954)	BASIC (1964)	Pascal (1970)	C++ (1983)	Java (1997)	C# (2000)
ALGOL 58 (1958)	LOGO (1968)	C (1971)	Eiffel (1986)		
LISP (1958)	Simula 67 (1967)	Modula 2 (1975)	Perl (1987)		
COBOL (1959)	Smalltalk (1969)	Ada (1979)			

Programación de la Web

Si después o en paralelo de su proceso de aprendizaje en fundamentos y metodología de la programación desea practicar no sólo con un lenguaje tradicional como Pascal, C, C++, Java o C#, sino introducirse en lenguajes de programación para la Web, enumeramos a continuación los más empleados en este campo.

Los programadores pueden utilizar una amplia variedad de lenguajes de programación, incluyendo C y C++ para escribir aplicaciones Web. Sin embargo, algunas herramientas de programación son, particularmente, útiles para desarrollar aplicaciones Web:

• HTML, técnicamente es un lenguaje de descripción de páginas más que un lenguaje de programación. Es el elemento clave para la programación en la Web.

• JavaScript, es un lenguaje interpretado de guionado (scripting) que facilita a los diseñadores de páginas Web añadir guiones a páginas Web y modos para enlazar esas páginas.

• VBScript, la respuesta de Microsoft a JavaScript basada en VisualBasic.

• Java, lenguaje de programación, por excelencia, de la Web.

• ActiveX, lenguaje de Microsoft para simular a algunas de las características de Java.

• C#, el verdadero competidor de Java y creado por Microsoft.

• Perl, lenguaje interpretado de guionado (scripting) idóneo para escritura de texto.

• XML, lenguaje de marcación que resuelve todas las limitaciones de HTML y ha sido el creador de una nueva forma de programar la Web. Es el otro gran lenguaje de la Web.

• AJAX, es el futuro de la Web. Es una mezcla de JavaScript y XML. Es la espina dorsal de la nueva generación Web 2.0.

1.5 Fases en la resolución de problemas.

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución en la misma. Aunque el proceso de diseñar programas es, esencialmente, un proceso creativo, se puede considerar una serie de fases o pasos comunes, que generalmente deben seguir todos los programadores.

Las fases de resolución de un problema con computadora son:

• Análisis del problema.
• Diseño del algoritmo.
• Codificación.
• Compilación y ejecución.
• Verificación.
• Depuración.
• Mantenimiento.
• Documentación.

Las características más sobresalientes de la resolución de problemas son:

• Análisis. El problema se analiza teniendo presente la especificación de los requisitos dados por el cliente de la empresa o por la persona que encarga el programa.

• Diseño. Una vez analizado el problema, se diseña una solución que conducirá a un algoritmo que resuelva el problema.

• Codificación (implementación). La solución se escribe en la sintaxis del lenguaje de alto nivel (por ejemplo, Pascal) y se obtiene un programa fuente que se compila a continuación.

• Ejecución, verificación y depuración. El programa se ejecuta, se comprueba rigurosamente y se eliminan todos los errores (denominados "bugs", en inglés) que puedan aparecer.

• Mantenimiento. El programa se actualiza y modifica, cada vez que sea necesario, de modo que se cumplan todas las necesidades de cambio de sus usuarios.

• Documentación. Escritura de las diferentes fases del ciclo de vida del software, esencialmente el análisis, diseño y codificación, unidos a manuales de usuario y de referencia, así como normas para el mantenimiento.

Las dos primeras fases conducen a un diseño detallado escrito en forma de algoritmo. Durante la tercera fase (codificación) se implementa1 el algoritmo en un código escrito en un lenguaje de programación, reflejando las ideas desarrolladas en las fases de análisis y diseño.

Las fases de compilación y ejecución traducen y ejecutan el programa. En las fases de verificación y depuración el programador busca errores de las etapas anteriores y los elimina. Comprobará que mientras más tiempo se gaste en la fase de análisis y diseño, menos se gastará en la depuración del programa. Por último, se debe realizar la documentación del programa.

Antes de conocer las tareas a realizar en cada fase, se considera el concepto y significado de la palabra algoritmo.

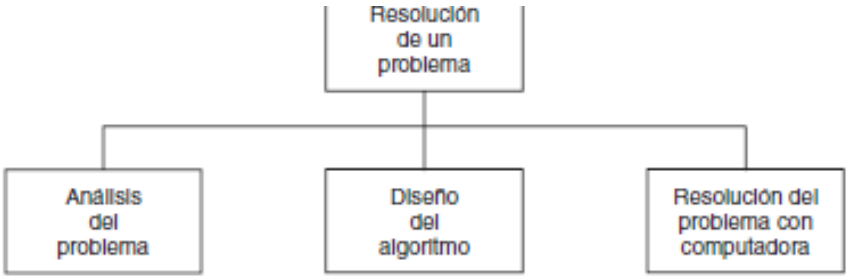
La palabra algoritmo se deriva de la traducción al latín de la palabra Alkhô-warîzmi2, nombre de un matemático y astrónomo árabe que escribió un tratado sobre manipulación de números y ecuaciones en el siglo IX. Un algoritmo es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos.

Un algoritmo debe producir un resultado en un tiempo finito. Los métodos que utilizan algoritmos se denominan métodos algorítmicos, en oposición a los métodos que implican algún juicio o interpretación que se denominan métodos heurísticos. Los métodos algorítmicos se pueden implementar en computadoras; sin embargo, los procesos heurísticos no han sido convertidos fácilmente en las computadoras. En los últimos años las técnicas de inteligencia artificial han hecho posible la implementación del proceso heurístico en computadoras.

Ejemplos de algoritmos son: instrucciones para montar en una bicicleta, hacer una receta de cocina, obtener el máximo común divisor de dos números, etc. Los algoritmos se pueden expresar por fórmulas, diagramas de flujo o N-S y pseudocódigos. Esta última representación es la más utilizada para su uso con lenguajes estructurados como Pascal.

1.6 Análisis del problema.

La primera fase de la resolución de un problema con computadora es el análisis del problema. Esta fase requiere una clara definición, donde se contemple exactamente lo que debe hacer el programa y el resultado o solución deseada. Dado que se busca una solución por computadora, se precisan especificaciones detalladas de entrada y salida. La Figura muestra los requisitos que se deben definir en el análisis.



Para poder identificar y definir bien un problema es conveniente responder a las siguientes preguntas:

- ¿Qué entradas se requieren? (tipo de datos con los cuales se trabaja y cantidad).
- ¿Cuál es la salida deseada? (tipo de datos de los resultados y cantidad).
- ¿Qué método produce la salida deseada?
- Requisitos o requerimientos adicionales y restricciones a la solución.

1.7 Diseño de algoritmo.

En la etapa de análisis del proceso de programación se determina qué hace el programa. En la etapa de diseño se determina cómo hace el programa la tarea solicitada. Los métodos más eficaces para el proceso de diseño se basan en el conocido divide y vencerás. Es decir, la resolución de un problema complejo se realiza dividiendo el problema en subproblemas

y a continuación dividiendo estos subproblemas en otros de nivel más bajo, hasta que pueda ser implementada una solución en la computadora. Este método se conoce técnicamente como diseño descendente (top-down) o modular. El proceso de romper el problema en cada etapa y expresar cada paso en forma más detallada se denomina refinamiento sucesivo.

Cada subprograma es resuelto mediante un módulo (subprograma) que tiene un solo punto de entrada y un solo punto de salida.

Cualquier programa bien diseñado consta de un programa principal (el módulo de nivel más alto) que llama a subprogramas (módulos de nivel más bajo) que a su vez pueden llamar a otros subprogramas. Los programas estructurados de esta forma se dice que tienen un diseño modular y el método de romper el programa en módulos más pequeños se llama programación modular. Los módulos pueden ser planeados, codificados, comprobados y depurados independientemente (incluso por diferentes programadores) y a continuación combinarlos entre sí. El proceso implica la ejecución de los siguientes pasos hasta que el programa se termina:

1. Programar un módulo.
2. Comprobar el módulo.
3. Si es necesario, depurar el módulo.
4. Combinar el módulo con los módulos anteriores.

El proceso que convierte los resultados del análisis del problema en un diseño modular con refinamientos sucesivos que permitan una posterior traducción a un lenguaje se denomina diseño del algoritmo.

El diseño del algoritmo es independiente del lenguaje de programación en el que se vaya a codificar posteriormente.

1.8 Herramientas de programación.

Las dos herramientas más utilizadas comúnmente para diseñar algoritmos son: diagramas de flujo y pseudocódigos.

Un diagrama de flujo (flowchart) es una representación gráfica de un algoritmo. Los símbolos utilizados han sido normalizados por el Instituto Norteamericano de Normalización (ANSI), y los más frecuentemente empleados se muestran en la Figura 2.2, junto con una plantilla utilizada para el dibujo de los diagramas de flujo (Figura 2.3). En la Figura 2.4 se representa el diagrama de flujo que resuelve el Problema 2.1.

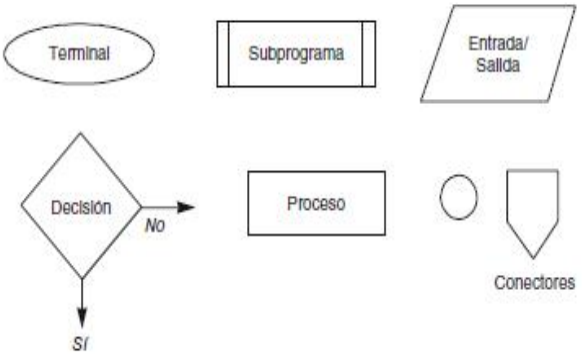
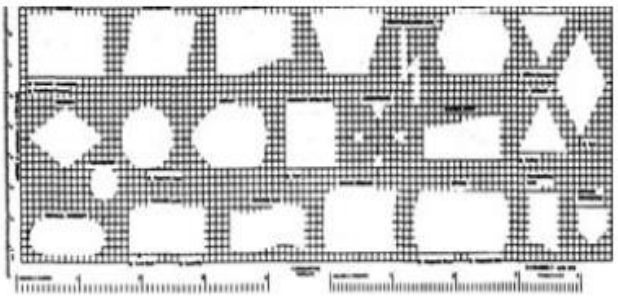


Figura 2.2. Símbolos más utilizados en los diagramas de flujo.



El pseudocódigo es una herramienta de programación en la que las instrucciones se escriben en palabras similares al inglés o español, que facilitan tanto la escritura como la lectura de programas. En esencia, el pseudocódigo se puede definir como un lenguaje de especificaciones de algoritmos.

Aunque no existen reglas para escritura del pseudocódigo en español, se ha recogido una notación estándar que se utilizará en el libro y que ya es muy empleada en los libros de programación en español3. Las palabras reservadas básicas se representarán en letras negritas minúsculas. Estas palabras son traducción libre de palabras reservadas de lenguajes como C, Pascal, etc. Más adelante se indicarán los pseudocódigos fundamentales para utilizar en esta obra.

1.9 Codificación de un programa.

La codificación es la escritura en un lenguaje de programación de la representación del algoritmo desarrollada en las etapas precedentes. Dado que el diseño de un algoritmo es independiente del lenguaje de programación utilizado para su implementación, el código puede ser escrito con igual facilidad en un lenguaje o en otro.

Para realizar la conversión del algoritmo en programa se deben sustituir las palabras reservadas en español por sus homónimos en inglés, y las operaciones/instrucciones indicadas en lenguaje natural por el lenguaje de programación correspondiente.

Documentación interna

Como se verá más tarde, la documentación de un programa se clasifica en interna y externa. La documentación interna es la que se incluye dentro del código del programa fuente mediante comentarios que ayudan a la comprensión del código. Todas las líneas de programas que comiencen con un símbolo / * son comentarios. El programa no los necesita y la computadora los ignora. Estas líneas de comentarios sólo sirven para hacer los programas más fáciles de comprender. El objetivo del programador debe ser escribir códigos sencillos y limpios.

Debido a que las máquinas actuales soportan grandes memorias (512 Mb o 1.024 Mb de memoria central mínima en computadoras personales) no es necesario recurrir a técnicas de ahorro de memoria, por lo que es recomendable que se incluya el mayor número de comentarios posibles, pero eso sí, que sean significativos.

1.10 Compilación y ejecución de un programa.

Una vez que el algoritmo se ha convertido en un programa fuente, es preciso introducirlo en memoria mediante el teclado y almacenarlo posteriormente en un disco. Esta operación se realiza con un programa editor. Posteriormente el programa fuente se convierte en un archivo de programa que se guarda (graba) en disco.

El programa fuente debe ser traducido a lenguaje máquina, este proceso se realiza con el compilador y el sistema operativo que se encarga prácticamente de la compilación.

Si tras la compilación se presentan errores (errores de compilación) en el programa fuente, es preciso volver a editar el programa, corregir los errores y compilar de nuevo. Este proceso se repite hasta que no se producen errores, obteniéndose el programa objeto que todavía no es ejecutable directamente. Suponiendo que no existen errores en el programa fuente, se debe instruir al sistema operativo para que realice la fase de montaje o enlace (link), carga, del programa objeto con las bibliotecas del programa del compilador. El proceso de montaje produce un programa ejecutable. La Figura 2.5 describe el proceso completo de compilación/ejecución de un programa.

Una vez que el programa ejecutable se ha creado, ya se puede ejecutar (correr o rodar) desde el sistema operativo con sólo teclear su nombre (en el caso de DOS). Suponiendo que no existen errores durante la ejecución (llamados errores en tiempo de ejecución), se obtendrá la salida de resultados del programa.

Las instrucciones u órdenes para compilar y ejecutar un programa en C, C++,... o cualquier otro lenguaje dependerá de su entorno de programación y del sistema operativo en que se ejecute Windows, Linux, Unix, etc.

1.11 Verificación y depuración de un programa.

La verificación o compilación de un programa es el proceso de ejecución del programa con una amplia variedad de datos de entrada, llamados datos de test o prueba, que determinarán si el programa tiene o no errores (“bugs”). Para realizar la verificación se debe desarrollar una amplia gama de datos de test: valores normales de entrada, valores extremos de entrada que comprueben los límites del programa y valores de entrada que comprueben aspectos especiales del programa.

La depuración es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores.

Cuando se ejecuta un programa, se pueden producir tres tipos de errores:

1. Errores de compilación. Se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación y suelen ser errores de sintaxis. Si existe un error de sintaxis,

la computadora no puede comprender la instrucción, no se obtendrá el programa objeto y el compilador imprimirá una lista de todos los errores encontrados durante la compilación.

2. Errores de ejecución. Estos errores se producen por instrucciones que la computadora puede comprender, pero no ejecutar. Ejemplos típicos son: división por cero y raíces cuadradas de números negativos. En estos casos se detiene la ejecución del programa y se imprime un mensaje de error.

3. Errores lógicos. Se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo. Estos errores son los más difíciles de detectar, ya que el programa puede funcionar y no producir errores de compilación ni de ejecución, y sólo puede advertirse el error por la obtención de resultados incorrectos. En este caso se debe volver a la fase de diseño del algoritmo, modificar el algoritmo, cambiar el programa fuente y compilar y ejecutar una vez más.

1.12 Documentación y mantenimiento.

problema. La importancia de la documentación debe ser destacada por su decisiva influencia en el producto final.

Programas pobremente documentados son difíciles de leer, más difíciles de depurar y casi imposibles de mantener y modificar.

La documentación de un programa puede ser interna y externa. La documentación interna es la contenida en líneas de comentarios. La documentación externa incluye análisis, diagramas de flujo y/o pseudocódigos, manuales de usuario con instrucciones para ejecutar el programa y para interpretar los resultados.

La documentación es vital cuando se desea corregir posibles errores futuros o bien cambiar el programa. Tales cambios se denominan mantenimiento del programa. Después de cada cambio la documentación debe ser actualizada para facilitar cambios posteriores. Es práctica frecuente numerar las sucesivas versiones de los programas 1.0, 1.1, 2.0, 2.1, etc. (Si los cambios introducidos son importantes, se varía el primer dígito [1.0, 2.0,...]; en caso de pequeños cambios sólo se varía el segundo dígito [2.0, 2.1…].)

UNIDAD II TIPOS DE PROGRAMACIÓN Y ALGORITMOS

2.1 Programación modular.

La programación modular es uno de los métodos de diseño más flexible y potente para mejorar la productividad de un programa. En programación modular el programa se divide en módulos (partes independientes), cada uno de los cuales ejecuta una única actividad o tarea y se codifican independientemente de otros módulos. Cada uno de estos módulos se analiza, codifica y pone a punto por separado. Cada programa contiene un módulo denominado programa principal que controla todo lo que sucede; se transfiere el control a submódulos (posteriormente se denominarán subprogramas), de modo que ellos puedan ejecutar sus funciones; sin embargo, cada submódulo devuelve el control al módulo principal cuando se haya completado su tarea. Si la tarea asignada a cada submódulo es demasiado compleja, éste deberá romperse en otros módulos más pequeños. El proceso sucesivo de subdivisión de módulos continúa hasta que cada módulo tenga solamente una tarea específica que ejecutar. Esta tarea puede ser entrada, salida, manipulación de datos, control de otros módulos o alguna combinación de éstos. Un módulo puede transferir temporalmente (bifurcar) el control a otro módulo; sin embargo, cada módulo debe eventualmente devolver el control al módulo del cual se recibe originalmente el control.

Los módulos son independientes en el sentido en que ningún módulo puede tener acceso directo a cualquier otro módulo excepto el módulo al que llama y sus propios submódulos. Sin embargo, los resultados producidos por un módulo pueden ser utilizados por cualquier otro módulo cuando se transfiera a ellos el control.

Dado que los módulos son independientes, diferentes programadores pueden trabajar simultáneamente en diferentes partes del mismo programa. Esto reducirá el tiempo del diseño del algoritmo y posterior codificación del programa.

Además, un módulo se puede modificar radicalmente sin afectar a otros módulos, incluso sin alterar su función principal.

La descomposición de un programa en módulos independientes más simples se conoce también como el método de divide y vencerás (divide and conquer). Cada módulo se diseña

con independencia de los demás, y siguiendo un método ascendente o descendente se llegará hasta la descomposición final del problema en módulos en forma jerárquica.

2.2 Programación estructurada.

C, Pascal, FORTRAN, y lenguajes similares, se conocen como lenguajes procedimentales (por procedimientos). Es decir, cada sentencia o instrucción señala al compilador para que realice alguna tarea: obtener una entrada, producir una salida, sumar tres números, dividir por cinco, etc. En resumen, un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias. En el caso de pequeños programas, estos principios de organización (denominados paradigma) se demuestran eficientes. El programador sólo tiene que crear esta lista de instrucciones en un lenguaje de programación, compilar en la computadora y ésta, a su vez, ejecuta estas instrucciones. Cuando los programas se vuelven más grandes, cosa que lógicamente sucede cuando aumenta la complejidad del problema a resolver, la lista de instrucciones aumenta considerablemente, de modo tal que el programador tiene muchas dificultades para controlar ese gran número de instrucciones. Los programadores pueden controlar, de modo normal, unos centenares de líneas de instrucciones. Para resolver este problema los programas se descompusieron en unidades más pequeñas que adoptaron el nombre de funciones (procedimientos, subprogramas o subrutinas en otros lenguajes de programación). De este modo en un programa orientado a procedimientos se divide en funciones, de modo que cada función tiene un propósito bien definido y resuelve una tarea concreta, y se diseña una interfaz claramente definida (el prototipo o cabecera de la función) para su comunicación con otras funciones.

Con el paso de los años, la idea de romper el programa en funciones fue evolucionando y se llegó al agrupamiento de las funciones en otras unidades más grandes llamadas módulos (normalmente, en el caso de C, denominadas archivos o ficheros); sin embargo, el principio seguía siendo el mismo: agrupar componentes que ejecutan listas de instrucciones (sentencias). Esta característica hace que a medida que los programas se hacen más grandes y complejos, el paradigma estructurado comienza a dar señales de debilidad y resultando muy difícil terminar los programas de un modo eficiente. Existen varias razones de la debilidad de los programas estructurados para resolver problemas complejos. Tal vez las

dos razones más evidentes son éstas. Primero, las funciones tienen acceso ilimitado a los datos globales. Segundo, las funciones inconexas y datos, fundamentos del paradigma procedimental proporcionan un modelo pobre del mundo real.

2.3 Programación orientada a objetos.

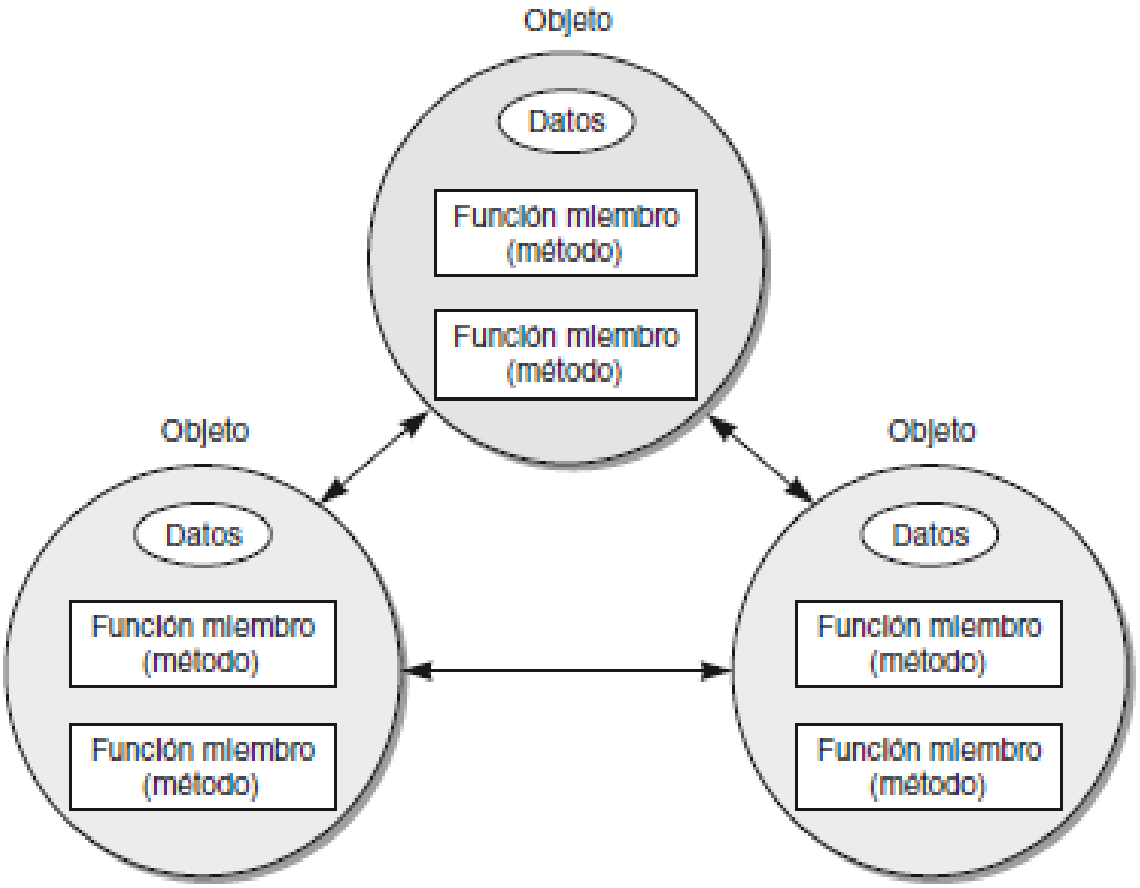
La programación orientada a objetos, tal vez el paradigma de programación más utilizado en el mundo del desarrollo de software y de la ingeniería de software del siglo XXI, trae un nuevo enfoque a los retos que se plantean en la programación estructurada cuando los problemas a resolver son complejos. Al contrario que la programación procedimental que enfatiza en los algoritmos, la POO enfatiza en los datos. En lugar de intentar ajustar un problema al enfoque procedimental de un lenguaje, POO intenta ajustar el lenguaje al problema. La idea es diseñar formatos de datos que se correspondan con las características esenciales de un problema.

La idea fundamental de los lenguajes orientados a objetos es combinar en una única unidad o módulo, tanto los datos como las funciones que operan sobre esos datos. Tal unidad se llama un objeto. Las funciones de un objeto se llaman funciones miembros en C++ o métodos (éste es el caso de Smalltalk, uno de los primeros lenguajes orientados a objetos), y son el único medio para acceder a sus datos. Los datos de un objeto, se conocen también como atributos o variables de instancia. Si se desea leer datos de un objeto, se llama a una función miembro del objeto. Se accede a los datos y se devuelve un valor. No se puede acceder a los datos directamente.

Los datos están ocultos, de modo que están protegidos de alteraciones accidentales. Los datos y las funciones se dice que están encapsulados en una única entidad. El encapsulamiento de datos y la ocultación de los datos son términos clave en la descripción de lenguajes orientados a objetos.

Si se desea modificar los datos de un objeto, se conoce exactamente cuáles son las funciones que interactúan con miembros del objeto. Ninguna otra función puede acceder a los datos. Esto simplifica la escritura, depuración y mantenimiento del programa. Un programa C++ se compone normalmente de un número de objetos que se comunican unos con otros

mediante la llamada a otras funciones miembro. La organización de un programa en C++ se muestra en la Figura 2.9. La llamada a una función miembro de un objeto se denomina enviar un mensaje a otro objeto.



2.4 Abstracción.

La abstracción es la propiedad de los objetos que consiste en tener en cuenta sólo los aspectos más importantes desde un punto de vista determinado y no tener en cuenta los restantes aspectos. El término abstracción que se suele utilizar en programación se refiere al hecho de diferenciar entre las propiedades externas de una entidad y los detalles de la composición interna de dicha entidad. Es la abstracción la que permite ignorar los detalles internos de un dispositivo complejo tal como una computadora, un automóvil, una lavadora o un horno de microondas, etc., y usarlo como una única unidad comprensible. Mediante

la abstracción se diseñan y fabrican estos sistemas complejos en primer lugar y, posteriormente, los componentes más pequeños de los cuales están compuestos. Cada componente representa un nivel de abstracción en el cual el uso del componente se aísla de los detalles de la composición interna del componente. La abstracción posee diversos grados denominados niveles de abstracción.

En consecuencia, la abstracción posee diversos grados de complejidad que se denominan niveles de abstracción que ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real. En el modelado orientado a objetos de un sistema esto significa centrarse en qué es y qué hace un objeto y no en cómo debe implementarse.

Durante el proceso de abstracción es cuando se decide qué características y comportamiento debe tener el modelo.

Aplicando la abstracción se es capaz de construir, analizar y gestionar sistemas de computadoras complejos y grandes que no se podrían diseñar si se tratara de modelar a un nivel detallado. En cada nivel de abstracción se visualiza el sistema en términos de componentes, denominados herramientas abstractas, cuya composición interna se ignora. Esto nos permite concentrarnos en cómo cada componente interactúa con otros componentes y centrarnos en la parte del sistema que es más relevante para la tarea a realizar en lugar de perderse a nivel de detalles menos significativos.

En estructuras o registros, las propiedades individuales de los objetos se pueden almacenar en los miembros. Para los objetos, no sólo es de interés cómo están organizados, sino también qué se puede hacer con ellos; es decir, las operaciones que forman la interfaz de un objeto son también importantes. El primer concepto en el mundo de la orientación a objetos nació con los tipos abstractos de datos (TAD). Un tipo abstracto de datos describe no sólo los atributos de un objeto, sino también su comportamiento (las operaciones). Esto puede incluir también una descripción de los estados que puede alcanzar un objeto.

Un medio de reducir la complejidad es la abstracción. Las características y los procesos se reducen a las propiedades esenciales, son resumidas o combinadas entre sí. De este modo, las características complejas se hacen más manejables.

2.5 Encapsulación y ocultación de datos.

El encapsulado o encapsulación de datos es el proceso de agrupar datos y operaciones relacionadas bajo la misma unidad de programación. En el caso de los objetos que poseen las mismas características y comportamiento se agrupan en clases, que no son más que unidades o módulos de programación que encapsulan datos y operaciones.

La ocultación de datos permite separar el aspecto de un componente, definido por su interfaz con el exterior, de sus detalles internos de implementación. Los términos ocultación de la información (information hiding) y encapsulación de datos (data encapsulation) se suelen utilizar como sinónimos, pero no siempre es así, y, muy al contrario,

son términos similares pero distintos. Normalmente, los datos internos están protegidos del exterior y no se puede acceder a ellos más que desde su propio interior y por tanto, no están ocultos. El acceso al objeto está restringido sólo a través de una interfaz bien definida.

El encapsulado o encapsulación de datos es el proceso de agrupar datos y operaciones relacionadas bajo la misma unidad de programación. En el caso de los objetos que poseen las mismas características y comportamiento se agrupan en clases, que no son más que unidades o módulos de programación que encapsulan datos y operaciones.

La ocultación de datos permite separar el aspecto de un componente, definido por su interfaz con el exterior, de sus detalles internos de implementación. Los términos ocultación de la información (information hiding) y encapsulación de datos (data encapsulation) se suelen utilizar como sinónimos, pero no siempre es así, y muy al contrario, son términos similares pero distintos. Normalmente, los datos internos están protegidos del exterior y no se puede acceder a ellos más que desde su propio interior y por tanto, no están ocultos. El acceso al objeto está restringido sólo a través de una interfaz bien definida.

2.6 Objetos.

El objeto es el centro de la programación orientada a objetos. Un objeto es algo que se visualiza, se utiliza y juega un rol o papel. Si se programa con enfoque orientado a objetos, se intentan descubrir e implementar los objetos que juegan un rol en el dominio del problema y en consecuencia programa. La estructura interna y el comportamiento de un objeto, en una primera fase, no tiene prioridad. Es importante que un objeto tal como un carro o una casa juegan un rol.

Dependiendo del problema, diferentes aspectos de un aspecto son relevantes. Un carro puede ser ensamblado de partes tales como un motor, una carrocería, unas puertas o puede ser descrito utilizando propiedades tales como su velocidad, su kilometraje o su fabricante. Estos atributos indican el objeto. De modo similar, una persona también se pue de ver como un objeto, del cual se disponen de diferentes atributos. Dependiendo de la definición del problema, esos atributos pueden ser el nombre, apellido, dirección, número de teléfono, color del cabello, altura, peso, profesión, etc.

Un objeto no necesariamente ha de realizar algo concreto o tangible. Puede ser totalmente abstracto y también puede describir un proceso. Por ejemplo, un partido de baloncesto o de rugby puede ser descrito como un objeto.

Los atributos de este objeto pueden ser los jugadores, el entrenador, la puntuación y el tiempo transcurrido de partido. Cuando se trata de resolver un problema con orientación a objetos, dicho problema no se descompone en funciones como en programación estructurada tradicional, caso de C, sino en objetos. El pensar en términos de objetos tiene una gran ventaja: se asocian los objetos del problema a los objetos del mundo real.

¿Qué tipos de cosas son objetos en los programas orientados a objetos? La respuesta está limitada por su imaginación, aunque se pueden agrupar en categorías típicas que facilitarán su búsqueda en la definición del problema de un modo más rápido y sencillo.

• Recursos Humanos:
— Empleados.
— Estudiantes.
— Clientes.
— Vendedores.
— Socios.

• Colecciones de datos:
— Arrays (arreglos).
— Listas.
— Pilas.
— Árboles.
— Árboles binarios.
— Grafos.
• Tipos de datos definidos por usuarios:
— Hora.
— Números complejos.
— Puntos del plano.
— Puntos del espacio.
— Ángulos.
— Lados.
• Elementos de computadoras:
— Menús.
— Ventanas.
— Objetos gráficos (rectángulos, círculos, rectas, puntos...).
— Ratón (mouse).
— Teclado.
— Impresora.
— USB.
— Tarjetas de memoria de cámaras fotográficas.
• Objetos físicos:
— Carros.
— Aviones.
— Trenes.
— Barcos.
— Motocicletas.
— Casas.
• Componentes de videojuegos:
— Consola.
— Mandos.
— Volante.
— Conectores.
— Memoria.
— Acceso a Internet.

La correspondencia entre objetos de programación y objetos del mundo real es el resultado eficiente de combinar datos y funciones que manipulan esos datos. Los objetos resultantes

ofrecen una mejor solución al diseño del programa que en el caso de los lenguajes orientados a procedimientos.

Un objeto se puede definir desde el punto de vista conceptual como una entidad individual de un sistema y que se caracteriza por un estado y un comportamiento. Desde el punto de vista de implementación un objeto es una entidad que posee un conjunto de datos y un conjunto de operaciones (funciones o métodos).

El estado de un objeto viene determinado por los valores que toman sus datos, cuyos valores pueden tener las restricciones impuestas en la definición del problema. Los datos se denominan también atributos y componen la estructura del objeto y las operaciones —también llamadas métodos— representan los servicios que proporciona el objeto.

2.7 Clases.

En POO los objetos son miembros de clases. En esencia, una clase es un tipo de datos al igual que cualquier otro tipo de dato definido en un lenguaje de programación. La diferencia reside en que la clase es un tipo de dato que contiene datos y funciones. Una clase contiene muchos objetos y es preciso definirla, aunque su definición no implica creación de objetos.

Una clase es, por consiguiente, una descripción de un número de objetos similares. Madonna, Sting, Prince, Juanes, Carlos Vives o Juan Luis Guerra son miembros u objetos de la clase "músicos de rock". Un objeto concreto, Juanes o Carlos Vives, son instancias de la clase "músicos de rock".

Una clase es una descripción general de un conjunto de objetos similares. Por definición todos los objetos de una clase comparten los mismos atributos (datos) y las mismas operaciones (métodos). Una clase encapsula las abstracciones de datos y operaciones necesarias para describir una entidad u objeto del mundo real.

Una clase se representa en UML mediante un rectángulo que contiene en una banda con el nombre de la clase y opcionalmente otras dos bandas con el nombre de sus atributos y de sus operaciones o métodos.

2.8 Generalización y especialización: herencia.

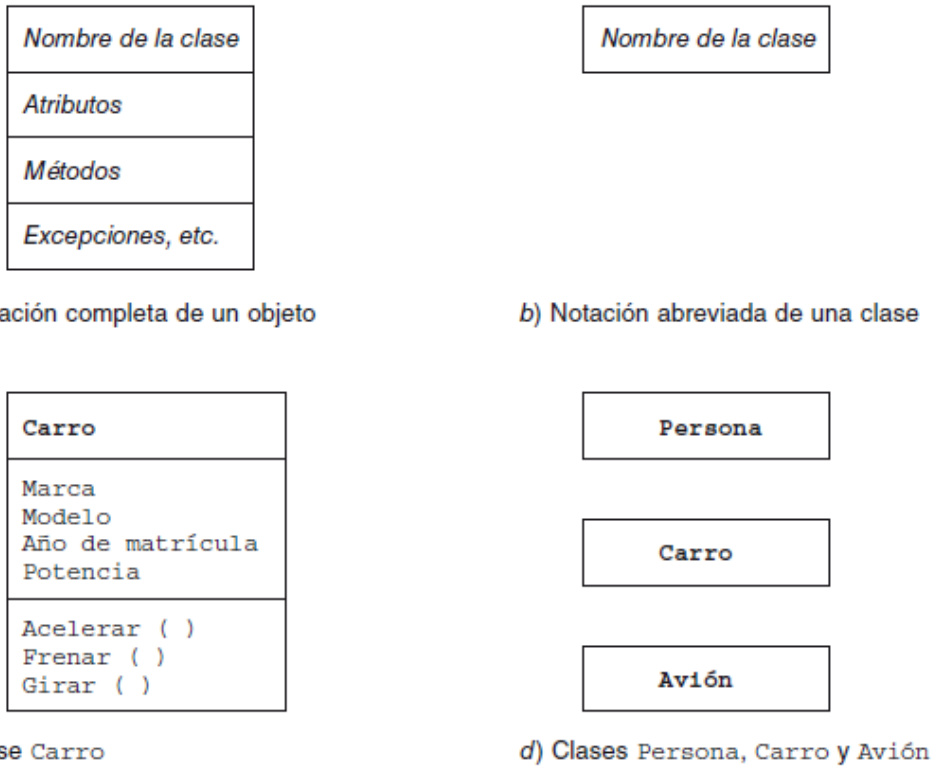
La generalización es la propiedad que permite compartir información entre dos entidades evitando la redundancia.

En el comportamiento de objetos existen con frecuencia propiedades que son comunes en diferentes objetos y esta propiedad se denomina generalización.

Por ejemplo, máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas, etc., son todos electrodomésticos (aparatos del hogar). En el mundo de la orientación a objetos, cada uno de estos aparatos es una subclase de la clase Electrodoméstico y a su vez Electrodoméstico es una superclase de todas las otras clases (máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas...).

El proceso inverso de la generalización por el cual se definen nuevas clases a partir de otras ya existentes se denomina especialización En orientación a objetos, el mecanismo que implementa la propiedad de generalización se denomina herencia.

La herencia permite definir nuevas clases a partir de otras clases ya existentes, de modo que presentan las mismas características y comportamiento de éstas, así como otras adicionales.



a) Notación completa de un objeto

b) Notación abreviada de una clase

c) Clase `Carro`

d) Clases `Persona`, `Carro` y `Avión`

Figura 2.11. Representación de clases en UML.

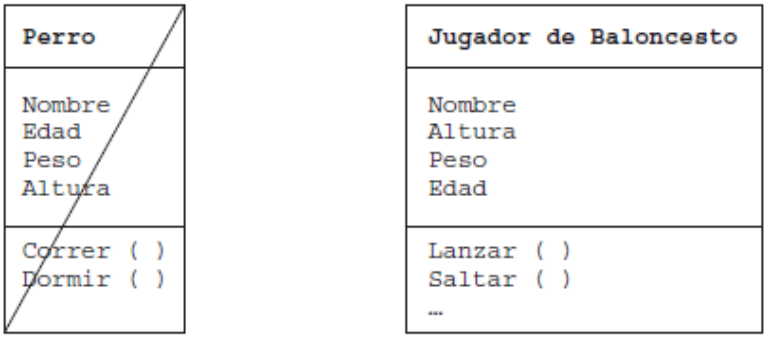


Figura 2.12. Representación de clases en UML con atributos y métodos.

La idea de clases conduce a la idea de herencia. Clases diferentes se pueden conectar unas con otras de modo jerárquico. Como ya se ha comentado anteriormente con las relaciones de generalización y especialización, en nuestras vidas diarias se utiliza el concepto de clases divididas en subclases. La clase animal se divide en anfibios, mamíferos, insectos, pájaros, etc., y la clase vehículo en carros, motos, camiones, buses, etc.

El principio de la división o clasificación es que cada subclase comparte características comunes con la clase de la que procede o se deriva. Los carros, motos, camiones y buses tienen ruedas, motores y carrocerías; son las características que definen a un vehículo. Además de las características comunes con los otros miembros de la clase, cada subclase tiene sus propias características. Por ejemplo, los camiones tienen una cabina independiente de la caja que transporta la carga; los buses tienen un gran número de asientos independientes para los viajeros que ha de transportar, etc. En la Figura 2.13 se muestran clases pertenecientes a una jerarquía o herencia de clases.

De modo similar una clase se puede convertir en padre o raíz de otras subclases. En C++ la clase original se denomina clase base y las clases que se derivan de ella se denominan clases derivadas y siempre son una especialización o concreción de su clase base. A la inversa, la clase base es la generalización de la clase derivada. Esto significa que todas las propiedades (atributos y operaciones) de la clase base se heredan por la clase derivada, normalmente suplementada con propiedades adicionales.

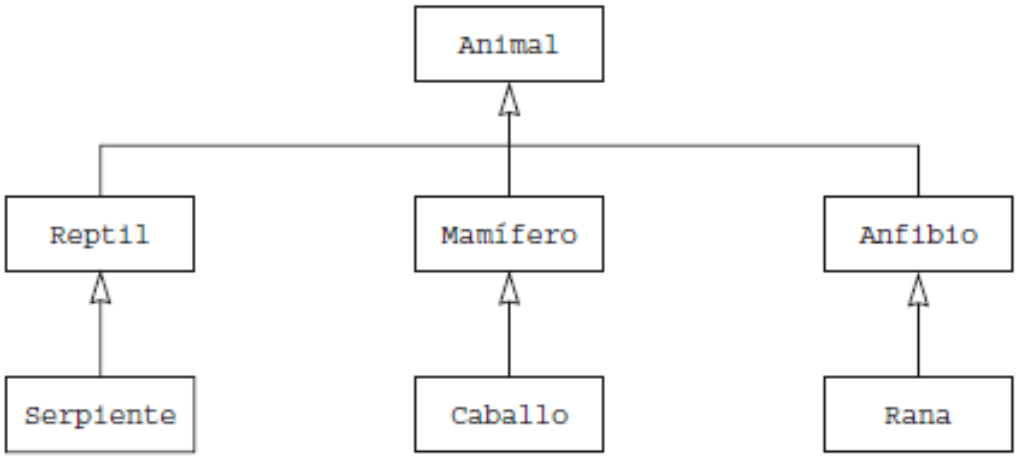


Figura 2.13. Herencia de clases en UML.

2.9 Reusabilidad.

Una vez que una clase ha sido escrita, creada y depurada, se puede distribuir a otros programadores para utilizar en sus propios programas. Esta propiedad se llama reusabilidad 4 o reutilización. Su concepto es similar a las funciones incluidas en las bibliotecas de

funciones de un lenguaje procedimental como C que se pueden incorporar en diferentes programas.

En C++, el concepto de herencia proporciona una extensión o ampliación al concepto de reusabilidad. Un programador puede considerar una clase existente y sin modificarla, añadir competencias y propiedades adicionales a ella. Esto se consigue derivando una nueva clase de una ya existente. La nueva clase heredará las características de la clase antigua, pero es libre de añadir nuevas características propias.

La facilidad de reutilizar o reusar el software existente es uno de los grandes beneficios de la POO: muchas empresas consiguen con la reutilización de clase en nuevos proyectos la reducción de los costes de inversión en sus presupuestos de programación. ¿En esencia cuáles son las ventajas de la herencia? Primero, se utiliza para consistencia y reducir código. Las propiedades comunes de varias clases sólo necesitan ser implementadas una vez y sólo necesitan modificarse una vez si es necesario. La otra ventaja es que el concepto de abstracción de la funcionalidad común está soportada.

2.10 Polimorfismo.

Además de las ventajas de consistencia y reducción de código, la herencia, aporta también otra gran ventaja: facilitar el polimorfismo. Polimorfismo es la propiedad de que un operador o una función actúen de modo diferente en función del objeto sobre el que se aplican. En la práctica, el polimorfismo significa la capacidad de una operación de ser interpretada sólo por el propio objeto que lo invoca. Desde un punto de vista práctico de ejecución del programa, el polimorfismo se realiza en tiempo de ejecución ya que durante la compilación no se conoce qué tipo de objeto y por consiguiente qué operación ha sido llamada. En el Capítulo 14 se describirá en profundidad la propiedad de polimorfismo y los diferentes modos de implementación del polimorfismo.

La propiedad de polimorfismo es aquella en que una operación tiene el mismo nombre en diferentes clases, pero se ejecuta de diferentes formas en cada clase. Así, por ejemplo, la operación de abrir se puede dar en diferentes clases: abrir una puerta, abrir una ventana, abrir un periódico, abrir un archivo, abrir una cuenta corriente en un banco, abrir un libro,

etc. En cada caso se ejecuta una operación diferente, aunque tiene el mismo nombre en todos ellos "abrir". El polimorfismo es la propiedad de una operación de ser interpretada sólo por el objeto al que pertenece.

Existen diferentes formas de implementar el polimorfismo y variará dependiendo del lenguaje de programación.

Veamos el concepto con ejemplos de la vida diaria.

En un taller de reparaciones de automóviles existen numerosos carros, de marcas diferentes, de modelos diferentes, de tipos diferentes, potencias diferentes, etc. Constituyen una clase o colección heterogénea de carros (coches).

Supongamos que se ha de realizar una operación común "cambiar los frenos del carro". La operación a realizar es la diferente, incluirá diferentes acciones en cada caso. Otro ejemplo a considerar y relativo a los operadores "+" y "*" aplicados a números enteros o números complejos; aunque ambos son números, en un caso la suma y multiplicación son operaciones simples, mientras que en el caso de los números complejos al componerse de parte real y parte imaginaria, será necesario seguir un método específico para tratar ambas partes y obtener un resultado que también será un número complejo.

El uso de operadores o funciones de forma diferente, dependiendo de los objetos sobre los que están actuando se llama polimorfismo (una cosa con diferentes formas). Sin embargo, cuando un operador existente, tal como + o =, se le permite la posibilidad de operar sobre nuevos tipos de datos, se dice entonces que el operador está sobrecargado.

La sobrecarga es un tipo de polimorfismo y una característica importante de la POO. En el Capítulo 10 se ampliará, también en profundidad, este nuevo concepto.

2.11 Algoritmos.

El objetivo fundamental de este texto es enseñar a resolver problemas mediante una computadora. El programador de computadora es antes que nada una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático. A lo largo de todo este libro nos

referiremos a la metodología necesaria para resolver problemas mediante programas, concepto que se denomina metodología de la programación. El eje central de esta metodología es el concepto, ya tratado, de algoritmo.

2.11.1 Concepto y características de los algoritmos.

Un algoritmo es un método para resolver un problema. Aunque la popularización del término ha llegado con el advenimiento de la era informática, algoritmo proviene —como se comentó anteriormente— de Mohammed al- KhoWârizmi, matemático persa que vivió durante el siglo IX y alcanzó gran reputación por el enunciado de las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales; la traducción al latín del apellido en la palabra algorismus derivó posteriormente en algoritmo. Euclides, el gran mate mático griego (del siglo IV a. C.) que inventó un método para encontrar el máximo común divisor de dos números, se considera con Al-Khowârizmi el otro gran padre de la algoritmia (ciencia que trata de los algoritmos).

El profesor Niklaus Wirth —inventor de Pascal, Modula-2 y Oberon— tituló uno de sus más famosos libros, Algoritmos + Estructuras de datos = Programas, significándonos que sólo se puede llegar a realizar un buen programa con el diseño de un algoritmo y una correcta estructura de datos. Esta ecuación será una de las hipótesis fundamentales consideradas en esta obra.

La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto.

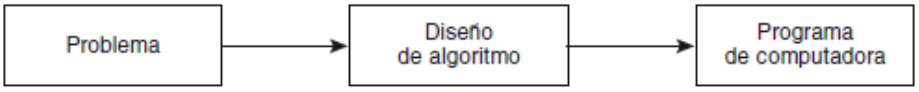


Figura 2.14. Resolución de un problema.

Los pasos para la resolución de un problema son:

1. Diseño del algoritmo, que describe la secuencia ordenada de pasos —sin ambigüedades— que conducen a lasolución de un problema dado. (Análisis del problema y desarrollo del algoritmo.)

2. Expresar el algoritmo como un programa en un lenguaje de programación adecuado. (Fase de codificación.)

3. Ejecución y validación del programa por la computadora.

Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo, de modo que sin algoritmo no puede existir un programa.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.

En la ciencia de la computación y en la programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

Dada la importancia del algoritmo en la ciencia de la computación, un aspecto muy importante será el diseño de algoritmos. A la enseñanza y práctica de esta tarea se dedica gran parte de este libro.

El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la programación. En esencia, la solución de un problema se puede expresar mediante un algoritmo.

Las características fundamentales que debe cumplir todo algoritmo son:

• Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.

• Un algoritmo debe estar bien definido. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.

• Un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: Entrada, Proceso y Salida. En el algoritmo de receta de cocina citado anteriormente se tendrá:

Entrada: Ingredientes y utensilios empleados.
Proceso: Elaboración de la receta en la cocina.
Salida: Terminación del plato (por ejemplo, cordero).

2.11.2 Diseño del algoritmo.

Una computadora no tiene capacidad para solucionar problemas más que cuando se le proporcionan los sucesivos pasos a realizar. Estos pasos sucesivos que indican las instrucciones a ejecutar por la máquina constituyen, como ya conocemos, el algoritmo.

La información proporcionada al algoritmo constituye su entrada y la información producida por el algoritmo constituye su salida.

Los problemas complejos se pueden resolver más eficazmente con la computadora cuando se rompen en subproblemas que sean más fáciles de solucionar que el original. Es el método de divide y vencerás (divide and conquer), mencionado anteriormente, y que consiste en dividir un problema complejo en otros más simples. Así, el problema de encontrar la superficie y la longitud de un círculo se puede dividir en tres problemas más simples o subproblemas (Figura 2.15).

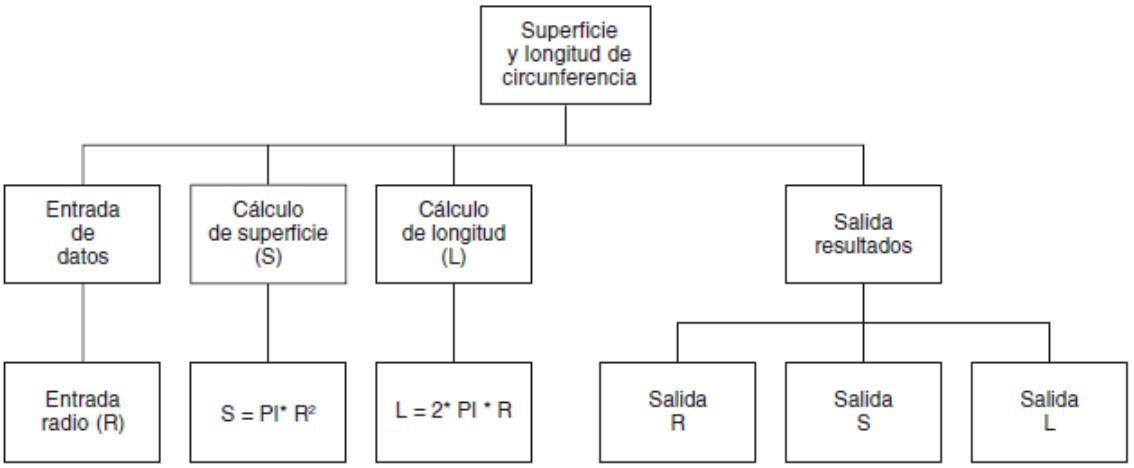


Figura 2.15. Refinamiento de un algoritmo.

La descomposición del problema original en subproblemas más simples y a continuación la división de estos subproblemas en otros más simples que pueden ser implementados para su solución en la computadora se denomina diseño descendente (top-down design). Normalmente, los pasos diseñados en el primer esbozo del algoritmo son incompletos e indicarán sólo unos pocos pasos (un máximo de doce aproximadamente). Tras esta primera descripción, éstos se amplían en una descripción más detallada con más pasos específicos. Este proceso se denomina refinamiento del algoritmo (stepwise refinement). Para problemas complejos se necesitan con frecuencia diferentes niveles de refinamiento antes de que se pueda obtener un algoritmo claro, preciso y completo.

El problema de cálculo de la circunferencia y superficie de un círculo se puede descomponer en subproblemas más simples: 1) leer datos de entrada; 2) calcular superficie y longitud de circunferencia, y 3) escribir resultados (datos de salida).

Las ventajas más importantes del diseño descendente son:

• El problema se comprende más fácilmente al dividirse en partes más simples denominadas módulos.
• Las modificaciones en los módulos son más fáciles.
• La comprobación del problema se puede verificar fácilmente.

Tras los pasos anteriores (diseño descendente y refinamiento por pasos) es preciso representar el algoritmo mediante una determinada herramienta de programación: diagrama de flujo, pseudocódigo o diagrama N-S.

2.11.3 Escritura de algoritmos.

Como ya se ha comentado anteriormente, el sistema para describir ("escribir") un algoritmo consiste en realizar una descripción paso a paso con un lenguaje natural del citado algoritmo. Recordemos que un algoritmo es un método o conjunto de reglas para solucionar un problema. En cálculos elementales estas reglas tienen las siguientes propiedades:

• deben ir seguidas de alguna secuencia definida de pasos hasta que se obtenga un resultado coherente,
• sólo puede ejecutarse una operación a la vez.

El flujo de control usual de un algoritmo es secuencial; consideremos el algoritmo que responde a la pregunta:

¿Qué hacer para ver la película de Harry Potter?

La respuesta es muy sencilla y puede ser descrita en forma de algoritmo general de modo similar a:

ir al cine
comprar una entrada (billete o ticket)
ver la película
regresar a casa

El algoritmo consta de cuatro acciones básicas, cada una de las cuales debe ser ejecutada antes de realizar la siguiente. En términos de computadora, cada acción se codificará en una o varias sentencias que ejecutan una tarea particular.

El algoritmo descrito es muy sencillo; sin embargo, como ya se ha indicado en párrafos anteriores, el algoritmo general se descompondrá en pasos más simples en un procedimiento denominado refinamiento sucesivo, ya que cada acción puede descomponerse a su vez en otras acciones simples. Así, por ejemplo, un primer refinamiento del algoritmo ir al cine se puede describir de la forma siguiente:

1. inicio
2. ver la cartelera de cines en el periódico
3. si no proyectan "Harry Potter" entonces

```
        3.1. decidir otra actividad
        3.2. bifurcar al paso 7
        si_no
        3.3. ir al cine
        fin_si
4. si hay cola entonces
        4.1. ponerse en ella
        4.2. mientras haya personas delante hacer
                4.2.1. avanzar en la cola
                fin_mientras
        fin_si
5. si hay localidades entonces
        5.1. comprar una entrada
        5.2. pasar a la sala
        5.3. localizar la(s) butaca(s)
        5.4. mientras proyectan la película hacer
                5.4.1. ver la película
                fin_mientras
        5.5. abandonar el cine
        si_no
        5.6. refunfuñar
        fin_si
6. volver a casa
7. fin
```

En el algoritmo anterior existen diferentes aspectos a considerar. En primer lugar, ciertas palabras reservadas se han escrito deliberadamente en negrita (mientras, si_no; etc.). Estas palabras describen las estructuras de control fundamentales y procesos de toma de decisión en el algoritmo. Éstas incluyen los conceptos importantes de selección (expresadas por si-entonces-si_no, if-then-else) y de repetición (expresadas con mientras-hacer o a veces repetir-hasta e iterar-fin_iterar, en inglés, while-do y repeat-until) que se encuentran en casi todos los algoritmos, especialmente en los de proceso de datos. La capacidad de decisión permite seleccionar alternativas de acciones a seguir o bien la repetición una y otra vez de operaciones básicas.

```
si proyectan la película seleccionada ir al cine
        si_no ver la televisión, ir al fútbol o leer el periódico
mientras haya personas en la cola, ir avanzando repetidamente
        hasta llegar a la taquilla
```

Otro aspecto a considerar es el método elegido para describir los algoritmos: empleo de indentación (sangrado o justificación) en escritura de algoritmos. En la actualidad es tan

importante la escritura de programa como su posterior lectura. Ello se facilita con la indentación de las acciones interiores a las estructuras fundamentales citadas: selectivas y repetitivas. A lo largo de todo el libro la indentación o sangrado de los algoritmos será norma constante.
Para terminar estas consideraciones iniciales sobre algoritmos, describiremos las acciones necesarias para refinar el algoritmo objeto de nuestro estudio; para ello analicemos la acción:

Localizar la(s) butaca(s).

Si los números de los asientos están impresos en la entrada, la acción compuesta se resuelve con el siguiente algoritmo:

```
1. inicio //algoritmo para encontrar la butaca del espectador
2. caminar hasta llegar a la primera fila de butacas
3. repetir
        compara número de fila con número impreso en billete
        si son iguales entonces pasar a la siguiente fila fin_si
        hasta_que se localice la fila correcta
4. mientras número de butaca no coincida con número de billete
        hacer avanzar a través de la fila a la siguiente butaca
        fin mientras
5. sentarse en la butaca
6. fin
```

En este algoritmo la repetición se ha mostrado de dos modos, utilizando ambas notaciones, repetir... hasta_ que y mientras... fin mientras. Se ha considerado también, como ocurre normalmente, que el número del asiento y fila coincide con el número y fila rotulado en el billete.

2.11.4 Representación gráfica de los algoritmos.

Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo del lenguaje de programación elegido. Ello permitirá que un algoritmo pueda ser codificado indistintamente en cualquier lenguaje.

Para conseguir este objetivo se precisa que el algoritmo sea representado gráfica o numéricamente, de modo que las sucesivas acciones no dependan de la sintaxis de ningún lenguaje de programación, sino que la descripción pueda servir fácilmente para su transformación en un programa, es decir, su codificación.

Los métodos usuales para representar un algoritmo son:

1. diagrama de flujo,
2. diagrama N-S (Nassi-Schneiderman),
3. lenguaje de especificación de algoritmos: pseudocódigo,
4. lenguaje español, inglés…
5. fórmulas.

Los métodos 4 y 5 no suelen ser fáciles de transformar en programas. Una descripción en español narrativo no es satisfactoria, ya que es demasiado prolija y generalmente ambigua. Una fórmula, sin embargo, es un buen sistema de representación. Por ejemplo, las fórmulas para la solución de una ecuación cuadrática (de segundo grado) son un medio sucinto de expresar el procedimiento algorítmico que se debe ejecutar para obtener las raíces de dicha ecuación.

xl = (–b + √b–2 – –4ac)/2a x2 = (–b – √b–2 – –4ac)/2ª y significa lo siguiente:

1. Elevar al cuadrado b.
2. Tomar a; multiplicar por c; multiplicar por 4.
3. Restar el resultado obtenido de 2 del resultado de 1, etc.

Sin embargo, no es frecuente que un algoritmo pueda ser expresado por medio de una simple fórmula.

2.11.5 Pseudocódigo.

El pseudocódigo es un lenguaje de especificación (descripción) de algoritmos. El uso de tal lenguaje hace el paso de codificación final (esto es, la traducción a un lenguaje de programación) relativamente fácil. Los lenguajes APL Pascal y Ada se utilizan a veces como lenguajes de especificación de algoritmos.

El pseudocódigo nació como un lenguaje similar al inglés y era un medio de representar básicamente las estructuras de control de programación estructurada que se verán en capítulos posteriores. Se considera un primer borrador, dado que el pseudocódigo tiene que traducirse posteriormente a un lenguaje de programación. El pseudocódigo no puede ser ejecutado por una computadora. La ventaja del pseudocódigo es que, en su uso, en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico. Es también fácil modificar el pseudocódigo si se descubren errores o anomalías en la lógica del

programa, mientras que en muchas ocasiones suele ser difícil el cambio en la lógica, una vez que está codificado en un lenguaje de programación. Otra ventaja del pseudocódigo es que puede ser traducido fácilmente a lenguajes estructurados como Pascal, C, FORTRAN 77/90, C++, Java, C#, etc.

El pseudocódigo original utiliza para representar las acciones sucesivas palabras reservadas en inglés —similares a sus homónimas en los lenguajes de programación—, tales como start, end, stop, if-then-else, while-end, repeat-until, etc. La escritura de pseudocódigo exige normalmente la indentación (sangría en el margen izquierdo) de diferentes líneas.

Una representación en pseudocódigo —en inglés— de un problema de cálculo del salario neto de un trabajador es la siguiente:

```
start
        //cálculo de impuesto y salarios
        read nombre, horas, precio
        salario ← horas * precio
        tasas ← 0,25 * salario
        salario_neto ← salario – tasas
        write nombre, salario, tasas, salario
end
```

El algoritmo comienza con la palabra start y finaliza con la palabra end, en inglés (en español, inicio, fin). Entre estas palabras, sólo se escribe una instrucción o acción por línea.

La línea precedida por // se denomina comentario. Es una información al lector del programa y no realiza ninguna instrucción ejecutable, sólo tiene efecto de documentación interna del programa. Algunos autores suelen utilizar corchetes o llaves.

No es recomendable el uso de apóstrofos o simples comillas como representan en algunos lenguajes primitivos los comentarios, ya que este carácter es representativo de apertura o cierre de cadenas de caracteres en lenguajes como Pascal o FORTRAN, y daría lugar a confusión.

Otro ejemplo aclaratorio en el uso del pseudocódigo podría ser un sencillo algoritmo del arranque matinal de un coche.

```
inicio
        //arranque matinal de un coche
        introducir la llave de contacto
        girar la llave de contacto
        pisar el acelerador
        oir el ruido del motor
        pisar de nuevo el acelerador
        esperar unos instantes a que se caliente el motor
fin
```

Por fortuna, aunque el pseudocódigo nació como un sustituto del lenguaje de programación y, por consiguiente, sus palabras reservadas se conservaron o fueron muy similares a las del idioma inglés, el uso del pseudocódigo se ha extendido en la comunidad hispana con términos en español como inicio, fin, parada, leer, escribir, si-entonces- si_no, mientras, fin_mientras, repetir, hasta_que, etc. Sin duda, el uso de la terminología del pseudocódigo en español ha facilitado y facilitará considerablemente el aprendizaje y uso diario de la programación.

En esta obra, al igual que en otras nuestras, utilizaremos el pseudocódigo en español y daremos en su momento las estructuras equivalentes en inglés, al objeto de facilitar la traducción del pseudocódigo al lenguaje de programación seleccionado.

Así pues, en los pseudocódigos citados anteriormente deberían ser sustituidas las palabras start, end, read, write, por inicio, fin, leer, escribir, respectivamente.

```
inicio start leer read
.
.
.
.
.
fin end escribir write
```

2.11.6 Diagrama de flujo.

Un diagrama de flujo (flowchart) es una de las técnicas de representación de algoritmos más antigua y a la vez más utilizada, aunque su empleo ha disminuido considerablemente, sobro todo, desde la aparición de lenguajes de programación estructurados. Un diagrama de flujo es un diagrama que utiliza los símbolos (cajas) estándar mostrados en la Tabla 2.1 y que tiene los pasos de algoritmo escritos en esas cajas unidas por flechas, denominadas líneas de flujo, que indican la secuencia en que se debe ejecutar.

La Figura 2.17 es un diagrama de flujo básico. Este diagrama representa la resolución de un programa que deduce el salario neto de un trabajador a partir de la lectura del nombre, horas trabajadas, precio de la hora, y sabiendo que los impuestos aplicados son el 25 por 100 sobre el salario bruto.

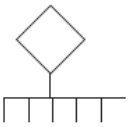
Los símbolos estándar normalizados por ANSI (abreviatura de American National Standars Institute) son muy variados. En la Figura 2.18 se representa una plantilla de dibujo típica donde se contemplan la mayoría de los símbolos utilizados en el diagrama; sin embargo, los símbolos más utilizados representan:

• proceso • decisión • conectores
• fin • entrada/salida • dirección del flujo

El diagrama de flujo de la Figura 2.17 resume sus características:

• existe una caja etiquetada “inicio”, que es de tipo elíptico,
• existe una caja etiquetada “fin” de igual forma que la anterior,
• si existen otras cajas, normalmente son rectangulares, tipo rombo o paralelogramo (el resto de las figuras se utilizan sólo en diagramas de flujo generales o de detalle y no siempre son imprescindibles).

Tabla 2.1. Símbolos de diagrama de flujo

Símbolos principales	Función
	Terminal (representa el comienzo, "inicio", y el final, "fin" de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en un programa.
	Entrada/Salida (cualquier tipo de introducción de datos en la memoria desde los periféricos, "entrada", o registro de la información procesada en un periférico, "salida".
	Proceso (cualquier tipo de operación que pueda originar cambio de valor, formato o posición de la información almacenada en memoria, operaciones aritméticas, de transferencia, etc.).
	Decisión (indica operaciones lógicas o de comparación entre datos —normalmente dos— y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa se debe seguir; normalmente tiene dos salidas —respuestas SÍ o NO— pero puede tener tres o más, según los casos).
	Decisión múltiple (en función del resultado de la comparación se seguirá uno de los diferentes caminos de acuerdo con dicho resultado).
	Conector (sirve para enlazar dos partes cualesquiera de un ordinograma a través de un conector en la salida y otro conector en la entrada. Se refiere a la conexión en la misma página del diagrama.
	Indicador de dirección o línea de flujo (indica el sentido de ejecución de las operaciones).
	Línea conectora (sirve de unión entre dos símbolos).
	Conector (conexión entre dos puntos del organigrama situado en páginas diferentes).
	Llamada a subrutina o a un proceso predeterminado (una subrutina es un módulo independientemente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal).
	Pantalla (se utiliza en ocasiones en lugar del símbolo de E/S).
	Impresora (se utiliza en ocasiones en lugar del símbolo de E/S).
	Teclado (se utiliza en ocasiones en lugar del símbolo de E/S).
	Comentarios (se utiliza para añadir comentarios clasificadores a otros símbolos del diagrama de flujo. Se pueden dibujar a cualquier lado del símbolo).

Problema:

Calcular el salario bruto y el salario neto de un trabajador "por horas" conociendo el nombre, número de horas trabajadas, impuestos a pagar y salario neto.

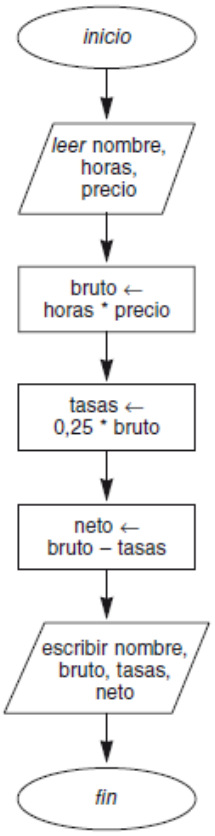


Figura 2.17. Diagrama de flujo.

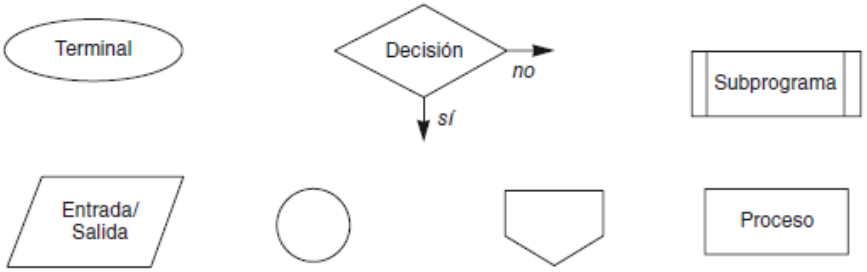


Figura 2.18. Plantilla típica para diagramas de flujo.

Se puede escribir más de un paso del algoritmo en una sola caja rectangular. El uso de flechas significa que la caja no necesita ser escrita debajo de su predecesora. Sin embargo, abusar demasiado de esta flexibilidad conduce a diagramas de flujo complicados e ininteligibles.

EJEMPLO 2.7

Calcular la media de una serie de números positivos, suponiendo que los datos se leen desde un terminal. Un valor de cero —como entrada— indicará que se ha alcanzado el final de la serie de números positivos.

El primer paso a dar en el desarrollo del algoritmo es descomponer el problema en una serie de pasos secuenciales. Para calcular una media se necesita sumar y contar los valores. Por consiguiente, nuestro algoritmo en forma descriptiva sería:

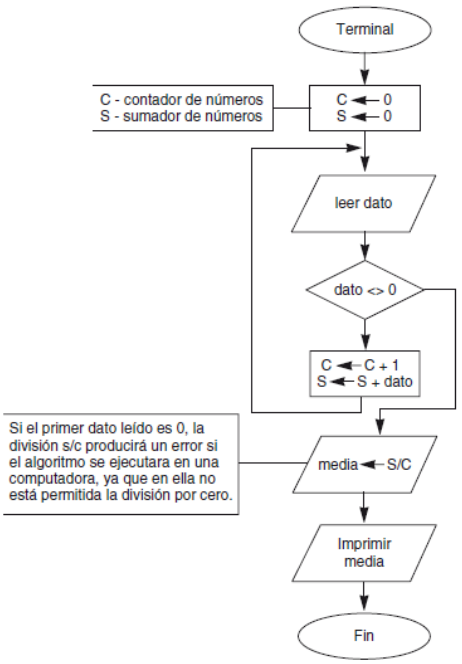
1. Inicializar contador de números C y variable suma S.
2. Leer un número.
3. Si el número leído es cero:
   - calcular la media;
   - imprimir la media;
   - fin del proceso.

   Si el número leído no es cero:
   - calcular la suma;
   - incrementar en uno el contador de números;
   - ir al paso 2.
4. Fin.

El refinamiento del algoritmo conduce a los pasos sucesivos necesarios para realizar las operaciones de lectura, verificación del último dato, suma y media de los datos.

Si el primer dato leído es 0, la división S/C produciría un error si el algoritmo se ejecutara en una computadora, ya que en ella no está permitida la división por cero.

Diagrama de flujo



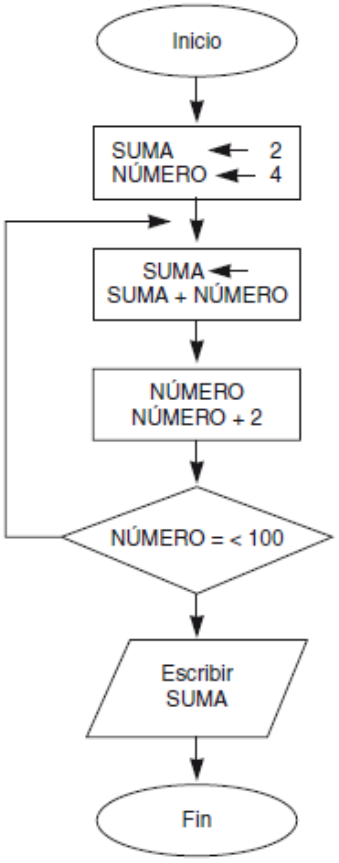
Pseudocódigo

```
entero: dato, C
Real: Media, S

C ← 0
S ← 0
escribir('Datos numéricos;
     para finalizar se introduce 0')

repetir
  leer(dato)
  si dato <> = 0 entonces
     C ← C + 1
     S ← S + dato
  fin_si

hasta dato = 0
{Calcula la media y la escribe}
si (C > 0) entonces
    Media ← S/C
    escribir (Media)
fin_si
```

EJEMPLO 2.8

Suma de los números pares comprendidos entre 2 y 100.

Diagrama de flujo



Pseudocódigo

```
entero:   numero,Suma
Suma ← 2
numero ← 4
mientras (numero <= 100) hacer
   suma ← suma + numero
   numero ← numero + 2
fin mientras
escribe ('Suma pares entre 2 y 100 =', suma)
```

UNIDAD III ESTRUCTURA GENERAL DE UN PROGRAMA

3.1 Concepto de programa.

Un programa de computadora es un conjunto de instrucciones —órdenes dadas a la máquina— que producirán la ejecución de una determinada tarea. En esencia, un programa es un medio para conseguir un fin. El fin será probablemente definido como la información necesaria para solucionar un problema.

El proceso de programación es, por consiguiente, un proceso de solución de problemas —como ya se vio en el anteriormente— y el desarrollo de un programa requiere las siguientes fases:

1. definición y análisis del problema;
2. diseño de algoritmos:
    - diagrama de flujo,
    - diagrama N-S,
    - pseudocódigo;
3. codificación del programa;
4. depuración y verificación del programa;
5. documentación;
6. mantenimiento.

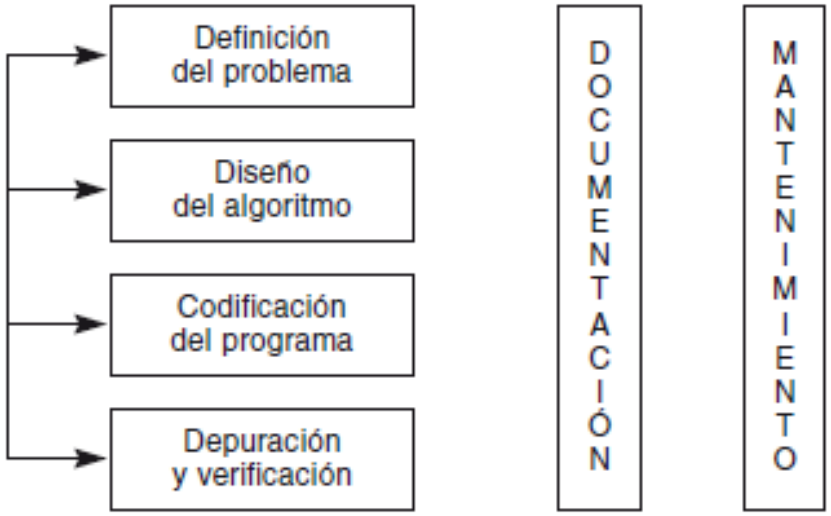


Figura 3.1. El proceso de la programación.

3.2 Partes constitutivas de un programa.

Tras la decisión de desarrollar un programa, el programador debe establecer el conjunto de especificaciones que debe contener el programa: entrada, salida y algoritmos de resolución, que incluirán las técnicas para obtener las salidas a partir de las entradas.

Conceptualmente un programa puede ser considerado como una caja negra, como se muestra en la Figura 3.2. La caja negra o el algoritmo de resolución, en realidad, es el conjunto de códigos que transforman las entradas del programa (datos) en salidas (resultados).

El programador debe establecer de dónde provienen las entradas al programa. Las entradas, en cualquier caso, procederán de un dispositivo de entrada —teclado, disco...—. El proceso de introducir la información de entrada —datos— en la memoria de la computadora se denomina entrada de datos, operación de lectura o acción de leer.

Las salidas de datos se deben presentar en dispositivos periféricos de salida: pantalla, impresoras, discos, etc. La operación de salida de datos se conoce también como escritura o acción de escribir.

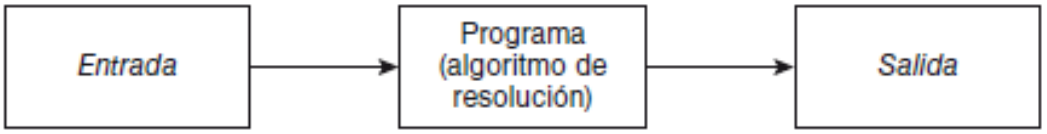


Figura 3.2. Bloques de un programa.

3.3 Instrucciones y tipos de instrucciones.

El proceso de diseño del algoritmo o posteriormente de codificación del programa consiste en definir las acciones o instrucciones que resolverán el problema.

Las acciones o instrucciones se deben escribir y posteriormente almacenar en memoria en el mismo orden en que han de ejecutarse, es decir, en secuencia.

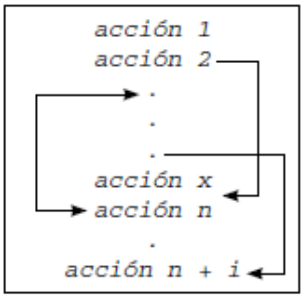
Un programa puede ser lineal o no lineal. Un programa es lineal si las instrucciones se ejecutan secuencialmente, sin bifurcaciones, decisión ni comparaciones.

```
instrucción 1
instrucción 2
      .
      .
      .
instrucción n
```

En el caso del algoritmo las instrucciones se suelen conocer como *acciones*, y se tendría:

```
acción 1
acción 2
   .
   .
   .
acción n
```

Un programa es *no lineal* cuando se interrumpe la secuencia mediante instrucciones de bifurcación.



Tipos de instrucciones

Las instrucciones disponibles en un lenguaje de programación dependen del tipo de lenguaje. Por ello, en este apartado estudiaremos las instrucciones —acciones— básicas que se pueden implementar de modo general en un algoritmo y que esencialmente soportan todos los lenguajes. Dicho de otro modo, las instrucciones básicas son independientes del lenguaje. La clasificación más usual, desde el punto de vista anterior, es:

1. instrucciones de inicio/fin,
2. instrucciones de asignación,
3. instrucciones de lectura,
4. instrucciones de escritura,
5. instrucciones de bifurcación.

Algunas de estas instrucciones se recogen en la Tabla 3.1.

Tabla 3.1. Instrucciones/acciones básicas

Tipo de instrucción	Pseudocódigo inglés	Pseudocódigo español
comienzo de proceso	`begin`	`inicio`
fin de proceso	`end`	`fin`
entrada (lectura)	`read`	`leer`
salida (escritura)	`write`	`escribir`
asignación	`A ← 5`	`B ← 7`

Instrucciones de asignación

Como ya son conocidas del lector, repasaremos su funcionamiento con ejemplos:

a) A ← 80 la variable A toma el valor de 80.

b) ¿Cuál será el valor que tomará la variable C tras la ejecución de las siguientes instrucciones?

A ← 12

B ← A

C ← B

A contiene 12, B contiene 12 y C contiene 12.

Nota

Antes de la ejecución de las tres instrucciones, el valor de A, B y C es indeterminado. Si se desea darles un valor inicial, habrá que hacerlo explícitamente, incluso cuando este valor sea 0. Es decir, habrá que definir e inicializar las instrucciones.

A ← 0

B ← 0

C ← 0

c) ¿Cuál es el valor de la variable AUX al ejecutarse la instrucción 5?

1. A ← 10

2. B ← 20

3. AUX ← A

4. A ← B

5. B ← AUX

• en la instrucción 1, A toma el valor 10

• en la instrucción 2, B toma el valor 20

• en la instrucción 3, AUX toma el valor anterior de A, o sea 10

• en la instrucción 4, A toma el valor anterior de B, o sea 20

• en la instrucción 5, B toma el valor anterior de AUX, o sea 10

• tras la instrucción 5, AUX sigue valiendo 10.

d) ¿Cuál es el significado de N ← N + 5 si N tiene el valor actual de 2?

N ← N + 5

Se realiza el cálculo de la expresión N + 5 y su resultado 2 + 5 = 7 se asigna a la variable situada a la izquierda, es decir, N tomará un nuevo valor 7.

Se debe pensar en la variable como en una posición de memoria, cuyo contenido puede variar mediante instrucciones de asignación (un símil suele ser un buzón de correos, donde el número de cartas depositadas en él variará según el movimiento diario del cartero de introducción de cartas o del dueño del buzón de extracción de dichas cartas).

Instrucciones de lectura de datos (entrada)

Esta instrucción lee datos de un dispositivo de entrada. ¿Cuál será el significado de las instrucciones siguientes?

a) leer (NÚMERO, HORAS, TASA)

Leer del terminal los valores NÚMERO, HORAS y TASAS, archivándolos en la memoria; si los tres números se teclean en respuesta a la instrucción son 12325, 32, 1200, significaría que se han asignado a las variables esos valores y equivaldría a la ejecución de las instrucciones.

NÚMERO ← 12325

HORAS ← 32

TASA ← 1200

b) leer (A, B, C)

Si se leen del terminal 100, 200, 300, se asignarían a las variables los siguientes valores:

A = 100

B = 200

C = 300

Instrucciones de escritura de resultados (salida)

Estas instrucciones se escriben en un dispositivo de salida. Explicar el resultado de la ejecución de las siguientes instrucciones:

A ← 100

B ← 200

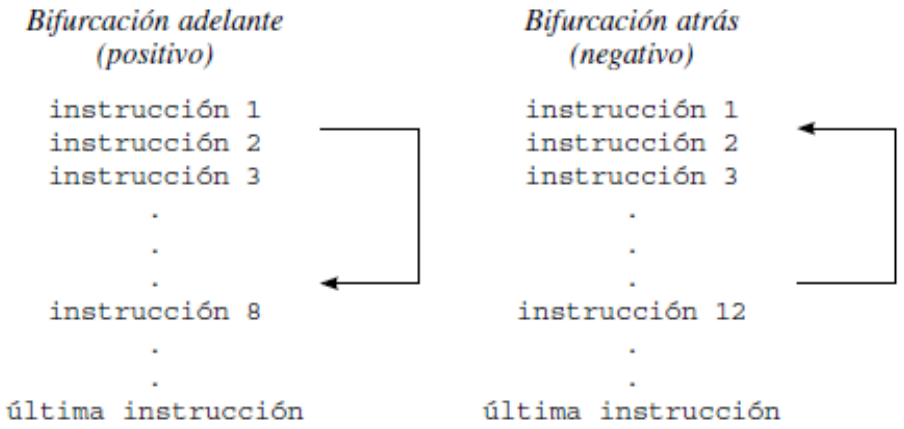
C ← 300

escribir (A, B, C)

Se visualizarían en la pantalla o imprimirían en la impresora los valores 100, 200 y 300 que contienen las variables A, B y C.

Instrucciones de bifurcación

El desarrollo lineal de un programa se interrumpe cuando se ejecuta una bifurcación. Las bifurcaciones pueden ser, según el punto del programa a donde se bifurca, hacia adelante o hacia atrás.



Las bifurcaciones en el flujo de un programa se realizarán de modo condicional en función del resultado de la evaluación de la condición.

Bifurcación incondicional: la bifurcación se realiza siempre que el flujo del programa pase por la instrucción sin necesidad del cumplimiento de ninguna condición (véase Figura 3.3).

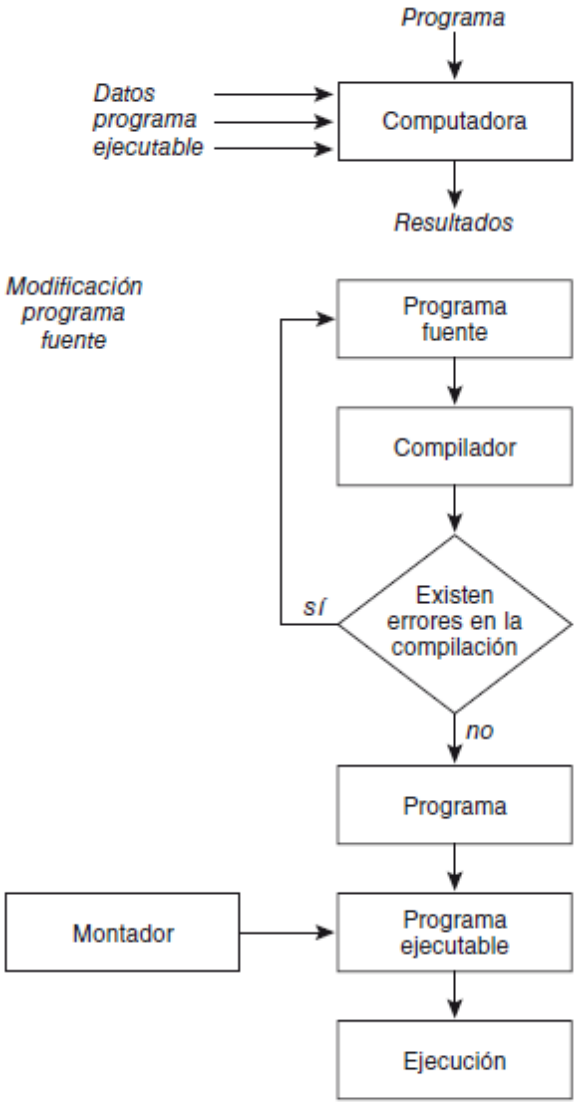


Figura 3.3. Fases de la ejecución de un programa.

Bifurcación condicional: la bifurcación depende del cumplimiento de una determinada condición. Si se cumple la condición, el flujo sigue ejecutando la acción F2. Si no se cumple, se ejecuta la acción F1 (véase Figura 3.4).

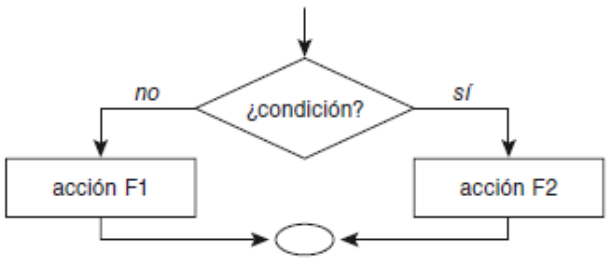


Figura 3.4. Bifurcación condicional.

3.4 Elementos básicos de un programa.

En programación se debe separar la diferencia entre el diseño del algoritmo y su implementación en un lenguaje específico. Por ello, se debe distinguir claramente entre los conceptos de programación y el medio en que ellos se implementan en un lenguaje específico. Sin embargo, una vez que se comprendan cómo utilizar los conceptos de programación y, la enseñanza de un nuevo lenguaje es relativamente fácil.

Los lenguajes de programación —como los restantes lenguajes— tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para las que esos elementos se combinan. Estas reglas se denominan sintaxis del lenguaje. Solamente las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora y los programas que contengan errores de sintaxis son rechazados por la máquina. Los elementos básicos constitutivos de un programa o algoritmo son:

• palabras reservadas (inicio, fin, si-entonces..., etc.),
• identificadores (nombres de variables esencialmente, procedimientos, funciones, nombre del programa, etc.),
• caracteres especiales (coma, apóstrofo, etc.),
• constantes,
• variables,
• expresiones,
• instrucciones.

Además de estos elementos básicos, existen otros elementos que forman parte de los programas, cuya comprensión y funcionamiento será vital para el correcto diseño de un algoritmo y naturalmente la codificación del programa.

Estos elementos son:

• bucles,
• contadores,
• acumuladores,
• interruptores,
• estructuras:

1. secuenciales,

2. selectivas,
3. repetitivas.

El amplio conocimiento de todos los elementos de programación y el modo de su integración en los programas constituyen las técnicas de programación que todo buen programador debe conocer.

3.5 Datos, tipos de datos y operaciones primitivas.

El primer objetivo de toda computadora es el manejo de la información o datos. Estos datos pueden ser las cifras de ventas de un supermercado o las calificaciones de una clase. Un dato es la expresión general que describe los objetos con los cuales opera una computadora. La mayoría de las computadoras pueden trabajar con varios tipos (modos) de datos. Los algoritmos y los programas correspondientes operan sobre esos tipos de datos.

La acción de las instrucciones ejecutables de las computadoras se refleja en cambios en los valores de las partidas de datos. Los datos de entrada se transforman por el programa, después de las etapas intermedias, en datos de salida.

En el proceso de resolución de problemas el diseño de la estructura de datos es tan importante como el diseño del algoritmo y del programa que se basa en el mismo.

Un programa de computadora opera sobre datos (almacenados internamente en la memoria almacenados en medios externos como discos, memorias USB, memorias de teléfonos celulares, etc., o bien introducidos desde un dispositivo como un teclado, un escáner o un sensor eléctrico). En los lenguajes de programación los datos deben de ser de un tipo de dato específico. El tipo de datos determina cómo se representan los datos en la computadora y los diferentes procesos que dicha computadora realiza con ellos.

Tipo de datos

Conjunto específico de valores de los datos y un conjunto de operaciones que actúan sobre esos datos.

Existen dos tipos de datos: básicos, incorporados o integrados (estándar) que se incluyen en los lenguajes de programación; definidos por el programador o por el usuario.

Además de los datos básicos o simples, se pueden construir otros datos a partir de éstos, y se obtienen los datos compuestos o datos agregados, tales como estructuras, uniones, enumeraciones (subrango, como caso particular de las enumeraciones, al igual de lo que sucede en Pascal), vectores o matrices/tablas y cadenas "arrays o arreglos"; también existen otros datos especiales en lenguajes como C y C++, denominados punteros (apuntadores) y referencias.

Existen dos tipos de datos: simples (sin estructura) y compuestos (estructurados). Los datos estructurados se estudian a partir del Capítulo 6 y son conjuntos de partidas de datos simples con relaciones definidas entre ellos.

Los distintos tipos de datos se representan en diferentes formas en la computadora. A nivel de máquina, un dato es un conjunto o secuencia de bits (dígitos 0 o 1). Los lenguajes de alto nivel permiten basarse en abstracciones e ignorar los detalles de la representación interna. Aparece el concepto de tipo de datos, así como su representación.

Los tipos de datos básicos son los siguientes:

numéricos (entero, real)
lógicos (boolean)
carácter (caracter, cadena)

Existen algunos lenguajes de programación —FORTRAN esencialmente— que admiten otros tipos de datos: complejos, que permiten tratar los números complejos, y otros lenguajes —Pascal— que también permiten declarar y definir sus propios tipos de datos: enumerados (enumerated) y subrango (subrange).

3.5.1 Datos numéricos.

El tipo numérico es el conjunto de los valores numéricos. Estos pueden representarse en dos formas distintas:

• tipo numérico entero (integer).

• tipo numérico real (real).

Enteros: el tipo entero es un subconjunto finito de los números enteros. Los enteros son números completos, no tienen componentes fraccionarios o decimales y pueden ser negativos o positivos. Ejemplos de números enteros son:

5 6
–15 4
20 17
1340 26

Los números enteros se pueden representar en 8, 16 o 32 bits, e incluso 64 bits, y eso da origen a una escala de enteros cuyos rangos dependen de cada máquina

Enteros	–32.768	a	32.767
Enteros cortos	–128	a	127
Enteros largos	–2147483648	a	2147483647

Además de los modificadores corto y largo, se pueden considerar sin signo (unsigned) y con signo (signed).

sin signo: 0 .. 65.5350
0 .. 4294967296

Los enteros se denominan en ocasiones números de punto o coma fija. Los números enteros máximos y mínimos de una computadora1 suelen ser –32.768 a +32.767. Los números enteros fuera de este rango no se suelen re presentar como enteros, sino como reales, aunque existen excepciones en los lenguajes de programación modernos como C, C++ y Java.

Reales: el tipo real consiste en un subconjunto de los números reales. Los números reales siempre tienen un punto decimal y pueden ser positivos o negativos. Un número real consta de un entero y una parte decimal. Los siguientes
ejemplos son números reales:

0.08	3739.41
3.7452	–52.321
–8.12	3.0

En aplicaciones científicas se requiere una representación especial para manejar números muy grandes, como la masa de la Tierra, o muy pequeños, como la masa de un electrón. Una computadora sólo puede representar un número fijo de dígitos. Este número puede variar de una máquina a otra, siendo ocho dígitos un número típico. Este límite provocará problemas para representar y almacenar números muy grandes o muy pequeños como son los ya citados o los siguientes:

4867213432 0.00000000387

Existe un tipo de representación denominado notación exponencial o científica y que se utiliza para números muy grandes o muy pequeños. Así,

367520100000000000000

se representa en notación científica descomponiéndolo en grupos de tres dígitos

367 520 100 000 000 000 000

y posteriormente en forma de potencias de 10

3.675201 x 1020

y de modo similar

.0000000000302579

se representa como

3.02579 x 10–11

La representación en coma flotante es una generalización de notación científica. Obsérvese que las siguientes expresiones son equivalentes:

3.675201 x 1019 = .3675207 x 1020 = .03675201 x 1021 = ...

= 36.75201 x 1018 = 367.5201 x 1017 = ...

En estas expresiones se considera la mantisa (parte decimal) al número real y el exponente (parte potencial) el de la potencia de diez.

36.75201 mantisa 18 exponente

Los tipos de datos reales se representan en coma o punto flotante y suelen ser de simple precisión, doble precisión o cuádruple precisión y suelen requerir 4 bytes, 8 bytes o 10-12 bytes, respectivamente. La Tabla 3.2 muestra los datos reales típicos en compiladores C/C++.

3.5.2 Datos lógicos (booleanos).

El tipo lógico —también denominado booleano— es aquel dato que sólo puede tomar uno de dos va lores:

cierto o verdadero (true) y falso (false).

Este tipo de datos se utiliza para representar las alternativas (sí/no) a determinadas condiciones. Por ejemplo, cuando se pide si un valor entero es par, la respuesta será verdadera o falsa, según sea par o impar.

C++ y Java soportan el tipo de dato bool.

3.5.3 Datos tipo carácter y tipo cadena.

El tipo carácter es el conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato tipo carácter contiene un solo carácter. Los caracteres que reconocen las diferentes computadoras no son estándar; sin embargo, la mayoría reconoce los siguientes caracteres alfabéticos y numéricos:

- caracteres alfabéticos (A, B, C, ..., Z) (a, b, c, ..., z),
- caracteres numéricos (1, 2, ..., 9, 0),
- caracteres especiales (+, -, *, /, ^, ., ;, <, >, $, ...).

Una cadena (string) de caracteres es una sucesión de caracteres que se encuentran delimitados por una comilla (apóstrofo) o dobles comillas, según el tipo de lenguaje de programación. La longitud de una cadena de caracteres es el número de ellos comprendidos entre los separadores o limitadores. Algunos lenguajes tienen datos tipo cadena.

'Hola Mortimer'
'12 de octubre de 1492'
'Sr. McKoy'

3.6 Constantes y variables.

Los programas de computadora contienen ciertos valores que no deben cambiar durante la ejecución del programa. Tales valores se llaman constantes. De igual forma, existen otros valores que cambiarán durante la ejecución del programa; a estos valores se les llama variables. Una constante es un dato que permanece sin cambios durante todo el desarrollo del algoritmo o durante la ejecución del programa.

Constantes reales válidas	Constantes reales no válidas
1.234	1,752.63 (comas no permitidas)
	82 (normalmente contienen un punto decimal, aunque existen lenguajes que lo admiten sin punto)
–0.1436	
+ 54437324	

Constantes reales en notación científica

3.374562E equivale a 3.374562 × 102

Una constante tipo carácter o constante de caracteres consiste en un carácter válido encerrado dentro de apóstrofos; por ejemplo,

'B' '+' '4' ';'

Si se desea incluir el apóstrofo en la cadena, entonces debe aparecer como un par de apóstrofos, encerrados dentro de simples comillas.

" "

Una secuencia de caracteres se denomina normalmente una cadena y una constante tipo cadena es una cadena encerrada entre apóstrofos. Por consiguiente,

'Juan Minguez'

y

'Pepe Luis Garcia'

son constantes de cadena válidas. Nuevamente, si un apóstrofo es uno de los caracteres en una constante de cadena, debe aparecer como un par de apóstrofos

'John"s'

Constantes lógicas (boolean)

Sólo existen dos constantes lógicas o boolean:

verdadero falso

La mayoría de los lenguajes de programación permiten diferentes tipos de constantes: enteras, rea les, caracteres y boolean o lógicas, y representan datos de esos tipos.

Una variable es un objeto o tipo de datos cuyo valor puede cambiar durante el desarrollo del algoritmo o ejecución del programa. Dependiendo del lenguaje, hay diferentes tipos de variables, tales como enteras, reales, carácter, lógicas y de cadena. Una variable que es de un cierto tipo puede tomar únicamente valores de ese tipo. Una variable de carácter, por ejemplo, puede tomar como valor sólo caracteres, mientras que una variable entera puede tomar sólo valores enteros.

Si se intenta asignar un valor de un tipo a una variable de otro tipo se producirá un error de tipo.

Una variable se identifica por los siguientes atributos: nombre que lo asigna y tipo que describe el uso de la variable.

Los nombres de las variables, a veces conocidos como identificadores, suelen constar de varios caracteres alfanuméricos, de los cuales el primero normalmente es una letra. No se deben utilizar —aunque lo permita el lenguaje, caso de FORTRAN— como nombres de identificadores palabras reservadas del lenguaje de programación. Nombres

válidos de variables son:

A510

NOMBRES	Letra	SalarioMes
NOTAS	Horas	SegundoApellido
NOMBRE_APELLIDOS2	Salario	Ciudad

Los nombres de las variables elegidas para el algoritmo o el programa deben ser significativos y tener relación con el objeto que representan, como pueden ser los casos siguientes:

NOMBRE para representar nombres de personas

PRECIOS para representar los precios de diferentes artículos

NOTAS para representar las notas de una clase

Existen lenguajes —Pascal— en los que es posible darles nombre a determinadas constantes típicas utilizadas en cálculos matemáticos, financieros, etc. Por ejemplo, las constantes π = 3.141592... y e = 2.718228 (base de los logaritmos naturales) se les pueden dar los nombres PI y E.

PI = 3.141592

E = 2.718228

Declaración de constantes y variables

Normalmente los identificadores de las variables y de las constantes con nombre deben ser declaradas en los programas antes de ser utilizadas. La sintaxis de la declaración de una variable suele ser:

<tipo_de_dato> <nombre_variable> [=<expresión>]

EJEMPLO

car letra, abreviatura

ent numAlumnos = 25

real salario = 23.000

Si se desea dar un nombre (identificador) y un valor a una constante de modo que su valor no se pueda modificar posteriormente, su sintaxis puede ser así:

const <tipo_de_dato> <nombre_constante> =<expresión>

EJEMPLO

const doble PI = 3.141592

const cad nombre = 'Mackoy'

const car letra = 'c'

3.7 Expresiones.

Las expresiones son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones especiales. Las mismas ideas son utilizadas en notación matemática tradicional; por ejemplo,

a + (b + 3) + √c

Aquí los paréntesis indican el orden de cálculo y √ representa la función raíz cuadrada.

Cada expresión toma un valor que se determina tomando los valores de las variables y constantes implicadas y la ejecución de las operaciones indicadas. Una expresión consta de operandos y operadores. Según sea el tipo de objetos que manipulan, las expresiones se clasifican en:

• aritméticas,
• relacionales,
• lógicas,
• carácter.

El resultado de la expresión aritmética es de tipo numérico; el resultado de la expresión relacional y de una expresiónlógica es de tipo lógico; el resultado de una expresión carácter es de tipo carácter.

3.7.1 Expresiones aritméticas.

Las expresiones aritméticas son análogas a las fórmulas matemáticas. Las variables y constantes son numéricas (real o entera) y las operaciones son las aritméticas.

+ suma

- resta

* multiplicación

/ división

↑, **, ^ exponenciación

div, / división entera

mod, % módulo (resto)

Los símbolos +, –, *, ^ (↑ o **) y las palabras clave div y mod se conocen como operadores aritméticos. En la expresión

5 + 3

los valores 5 y 3 se denominan operandos. El valor de la expresión 5 + 3 se conoce como resultado de la expresión.

Los operadores se utilizan de igual forma que en matemáticas. Por consiguiente, A · B se escribe en un algoritmo como A * B y 1/4 · C como C/4. Al igual que en matemáticas el signo menos juega un doble papel, como resta en A – B y como negación en –A.

*Todos los operadores aritméticos no existen en todos los lenguajes de programación; por ejemplo, en FORTRAN no existe div y mod. El operador exponenciación es diferente según sea el tipo de lenguaje de programación elegido (^, ↑ en BASIC, ** en FORTRAN).*

Los cálculos que implican tipos de datos reales y enteros suelen dar normalmente resultados del mismo tipo si los operandos lo son también. Por ejemplo, el producto de operandos reales produce un real (véase Tabla 3.3).

EJEMPLO

5 x 7 se representa por 5 * 7

6

4

se representa por 6/4

37 se representa por 3^7

Tabla 3.3. Operadores aritméticos

Operador	Significado	Tipos de operandos	Tipo de resultado
+	Signo positivo	Entero o real	Entero o real
-	Signo negativo	Entero o real	Entero o real
*	Multiplicación	Entero o real	Entero o real
/	División	Real	Real
div, /	División entera	Entero	Entero
mod, %	Módulo (resto)	Entero	Entero
++	Incremento	Entero	Entero
--	Decremento	Entero	Entero

Operadores DIV (/) y MOD (%)

El símbolo / se utiliza para la división real y la división entera (el operador div —en algunos lenguajes, por ejemplo BASIC, se suele utilizar el símbolo \— representa la división entera). El operador mod representa el resto de la división entera, y la mayoría de lenguajes utilizan el símbolo %.

A div B

Sólo se puede utilizar si A y B son expresiones enteras y obtiene la parte entera de A/B. Por consiguiente,

19 div 6 19/6

toma el valor 3. Otro ejemplo puede ser la división 15/6

En forma de operadores resultará la operación anterior

15 div 6 = 2 15 mod 6 = 3

Otros ejemplos son:

19 div 3 equivale a 6

19 mod 6 equivale a 1

EJEMPLO 3.1

Los siguientes ejemplos muestran resultados de expresiones aritméticas:

expresión	resultado	expresión	resultado
10.5/3.0	3.5	10/3	3
1/4	0.25	18/2	9
2.0/4.0	0.5	30/30	1
6/1	6.0	6/8	0
30/30	1.0	10%3	1
6/8	0.75	10%2	0

Operadores de incremento y decremento

Los lenguajes de programación C/C++, Java y C# soportan los operadores unitarios (unarios) de incremento, ++, y decremento, --. El operador de incremento (++) aumenta el valor de su operando en una unidad, y el operador de decremento (--) disminuye también en una unidad. El valor resultante dependerá de que el operador se emplee como prefijo o como sufijo (antes o después de la variable). Si actúa como prefijo, el operador cambia el valor de la variable y devuelve este nuevo valor; en caso contrario, si actúa como sufijo, el resultado de la expresión es el valor de la variable, y después se modifica esta variable.

++i Incrementa i en 1 y después utiliza el valor de i en la correspondiente expresión.

i++ Utiliza el valor de i en la expresión en que se encuentra y después se incrementa en 1.

--i Decrementa i en 1 y después utiliza el nuevo valor de i en la correspondiente expresión.

i-- Utiliza el valor de i en la expresión en que se encuentra y después se incrementa en 1.

EJEMPLO:

```
n = 5
escribir n
escribir n++
escribir n
n = 5
escribir n
escribir ++n
escribir n
```

Al ejecutarse el algoritmo se obtendría:

```
5
5
6
5
6
6
```

3.7.2 Reglas de prioridad.

Las expresiones que tienen dos o más operandos requieren unas reglas matemáticas que permitan determinar el orden de las operaciones, se denominan reglas de prioridad o precedencia y son:

1. Las operaciones que están encerradas entre paréntesis se evalúan primero. Si existen diferentes paréntesis anidados (interiores unos a otros), las expresiones más internas se evalúan primero.

2. Las operaciones aritméticas dentro de una expresión suelen seguir el siguiente orden de prioridad:

- operador ( )
- operadores ++, – – + y – unitarios,
- operadores *, /, % (producto, división, módulo)
- operadores +, – (suma y resta).

En los lenguajes que soportan la operación de exponenciación, este operador tiene la mayor prioridad.

En caso de coincidir varios operadores de igual prioridad en una expresión o subexpresión encerrada entre paréntesis, el orden de prioridad en este caso es de izquierda a derecha, y a esta propiedad se denomina asociatividad.

3.7.3 Expresiones lógicas (booleanas)

Un segundo tipo de expresiones es la expresión lógica o booleana, cuyo valor es siempre verdadero o falso. Recuerde que existen dos constantes lógicas, verdadera (true) y falsa (false) y que las variables lógicas pueden tomar sólo estos dos valores. En esencia, una expresión lógica es una expresión que sólo puede tomar estos dos valores, verdadero y falso. Se denominan también expresiones booleanas en honor del matemático británico George Boole, que desarrolló el Álgebra lógica de Boole.

Las expresiones lógicas se forman combinando constantes lógicas, variables lógicas y otras expresiones lógicas, utilizando los operadores lógicos not, and y or y los operadores relacionales (de relación o comparación) =, <, >, <=, >=, <>.

Operadores de relación

Los operadores relacionales o de relación permiten realizar comparaciones de valores de tipo numérico o carácter.

Los operadores de relación sirven para expresar las condiciones en los algoritmos. Los operadores de relación se recogen en la Tabla 3.4. El formato general para las comparaciones es

expresión1 operador de relación expresión2

y el resultado de la operación será verdadero o falso. Así, por ejemplo, si A = 4 y B = 3, entonces

A > B es verdadero

Tabla 3.4. Operadores de relación

Operador	Significado
<	menor que
>	mayor que
=, ==	igual que
<=	menor o igual que
>=	mayor o igual que
<>, !=	distinto de

mientras que

(A – 2) < (B – 4) es falso.

Los operadores de relación se pueden aplicar a cualquiera de los cuatro tipos de datos estándar: enteros, real, lógico, carácter. La aplicación a valores numéricos es evidente. Los ejemplos siguientes son significativos:

Para realizar comparaciones de datos tipo carácter, se requiere una secuencia de ordenación de los caracteres similar al orden creciente o decreciente. Esta ordenación suele ser alfabética, tanto mayúsculas como minúsculas, y numérica, considerándolas de modo independiente. Pero si se consideran caracteres mixtos, se debe recurrir a un código normalizado como es el ASCII (véase Apéndice A). Aunque no todas las computadoras siguen el código normalizado en su juego completo de caracteres, sí son prácticamente estándar los códigos de los caracteres alfanuméricos

más usuales. Estos códigos normalizados son:

• Los caracteres especiales #, %, $, (,), +, –, /, ..., exigen la consulta del código de ordenación.

• Los valores de los caracteres que representan a los dígitos están en su orden natural. Esto es, '0'<'1', '1'<'2', ..., '8'<'9'.

• Las letras mayúsculas A a Z siguen el orden alfabético ('A'<'B', 'C'<'F', etc.).

• Si existen letras minúsculas, éstas siguen el mismo criterio alfabético ('a'<'b', 'c'<'h', etc.).

En general, los cuatro grupos anteriores están situados en el código ASCII en orden creciente. Así, '1'<'A' y 'B'<'C'. Sin embargo, para tener completa seguridad será preciso consultar el código de caracteres de su computadora (normalmente, el ASCII, American Standar Code for Information Interchange o bien el ambiguo código EBCDIC, Extended Binary-Coded Decimal Interchange Code, utilizado en computadoras IBM diferentes a los modelos PC y PS/2).

Cuando se utilizan los operadores de relación, con valores lógicos, la constante false (falsa) es menor que la constante true (verdadera).

false < true

true > false

Si se utilizan los operadores relacionales = y <> para comparar cantidades numéricas, es importante recordar que la mayoría de los valores reales no pueden ser almacenados exactamente. En consecuencia, las expresiones lógicas

formales con comparación de cantidades reales con (=), a veces se evalúan como falsas, incluso aunque estas cantidades sean algebraicamente iguales. Así,

(1.0 / 3.0) * 3.0 = 1.0

teóricamente es verdadera y, sin embargo, al realizar el cálculo en una computadora se puede obtener .999999... y, en consecuencia, el resultado es falso; esto es debido a la precisión limitada de la aritmética real en las computadoras. Por consiguiente, a veces deberá excluir las comparaciones con datos de tipo real.

Operadores lógicos

Los operadores lógicos o booleanos básicos son not (no), and (y) y or (o). La Tabla 3.5 recoge el funcionamiento de dichos operadores.

Tabla 3.5. Operadores lógicos

Operador lógico	Expresión lógica	Significado
`no (not), !`	`no p (not p)`	negación de *p*
`y (and), &&`	`p y q (p and q)`	conjunción de *p* y *q*
`o (o), ||`	`p o q (p o q)`	disyunción de *p* y *q*

Las definiciones de las operaciones no, y, o se resumen en unas tablas conocidas como tablas de verdad.

a	no a	
verdadero	falso	no (6>10) es verdadera
falso	verdadero	ya que (6>10) es falsa.

a	b	a y b	
verdadero	verdadero	verdadero	a y b es verdadera sólo
verdadero	falso	falso	si a y b son verdaderas.
falso	verdadero	falso	
falso	falso	falso	

a	b	a o b	
verdadero	verdadero	verdadero	a o b es verdadera cuando
verdadero	falso	verdadero	a, b o ambas son verdaderas.
falso	verdadero	verdadero	
falso	falso	falso	

En las expresiones lógicas se pueden mezclar operadores de relación y lógicos. Así, por ejemplo,

```
(1 < 5) y (5 < 10)        es verdadera
(5 > 10) o ('A' < 'B')    es verdadera, ya que 'A' < 'B'
```

EJEMPLO 3.7

La Tabla 3.6 resume una serie de aplicaciones de expresiones lógicas.

Tabla 3.6. Aplicaciones de expresiones lógicas

Expresión lógica	Resultado	Observaciones
(1 > 0) y (3 = 3)	verdadero	
no PRUEBA	verdadero	· PRUEBA es un valor lógico falso.
(0 < 5) o (0 > 5)	verdadero	
(5 <= 7) y (2 > 4)	falso	
no (5 <> 5)	verdadero	
(numero = 1) o (7 >= 4)	verdadero	· numero es una variable entera de valor 5.

Prioridad de los operadores lógicos

Los operadores aritméticos seguían un orden específico de prioridad cuando existía más de un operador en las expresiones. De modo similar, los operadores lógicos y relaciones tienen un orden de prioridad.

Tabla 3.7. Prioridad de operadores (lenguaje Pascal)

Operador	Prioridad
`no` *(not)*	más alta (primera ejecutada).
`/, *, div, mod, y` *(and)*	↓
`+, -, o` *(or)*	↓
`<, >, =, <=, >=, <>`	más baja (última ejecutada).

Tabla 3.8. Prioridad de operadores (lenguajes C, C++, C# y Java)

Operador	Prioridad
++ y -- (incremento y decremento en 1), +, –, !	más alta
*, /, % (módulo de la división entera)	↓
+, - (suma, resta)	↓
<, <=, >, >=	↓
== (igual a), != (no igual a)	↓
&& (y lógica, AND)	↓
|| (o lógica, or)	↓
=, +=, -=, *=, /=, %= (operadores de asignación)	más baja

Al igual que en las expresiones aritméticas, los paréntesis se pueden utilizar y tendrán prioridad sobre cualquier operación.

EJEMPLO 3.8

`no 4 > 6`	produce un error, ya que el operador **no** se aplica a 4
`no (4 > 14)`	produce un valor verdadero
`(1.0 < x) y (x < z + 7.0)`	si x vale 7 y z vale 4, se obtiene un valor verdadero

3.8 La operación de asignación.

La operación de asignación es el modo de almacenar valores a una variable. La operación de asignación se representa con el símbolo u operador ← (en la mayoría de los lenguajes de programación, como C, C++, Java, el signo de la operación asignación es =). La operación de asignación se conoce como instrucción o sentencia de asignación cuando se refiere a un lenguaje de programación. El formato general de una operación de asignación es

<nombre de la variable> ← <expresión>

expresión es igual a expresión, variable o constante

La flecha (operador de asignación) se sustituye en otros lenguajes por = (Visual Basic, FORTRAN), := (Pascal) o = (Java, C++, C#). Sin embargo, es preferible el uso de la flecha en la redacción del algoritmo para evitar ambigüedades, dejando el uso del símbolo = exclusivamente para el operador de igualdad.

La operación de asignación:

A ← 5

significa que a la variable A se le ha asignado el valor 5.

La acción de asignar es destructiva, ya que el valor que tuviera la variable antes de la asignación se pierde y se reemplaza por el nuevo valor. Así, en la secuencia de operaciones

A ← 25
A ← 134
A ← 5

cuando éstas se ejecutan, el valor último que toma A será 5 (los valores 25 y 134 han desaparecido). La computadora ejecuta la sentencia de asignación en dos pasos. En el primero de ellos, el valor de la expresión al lado derecho del operador se calcula, obteniéndose un valor de un tipo específico. En el segundo caso, este valor se almacena en la variable cuyo nombre aparece a la izquierda del operador de asignación, sustituyendo al valor que tenía anteriormente.

X ← Y + 2

el valor de la expresión Y + 2 se asigna a la variable X.

Es posible utilizar el mismo nombre de variable en ambos lados del operador de asignación. Por ello, acciones como

N ← N + 1

tienen sentido; se determina el valor actual de la variable N, se incrementa en 1 y a continuación el resultado se asigna a la misma variable N. Sin embargo, desde el punto de vista matemático no tiene sentido N ← N + 1.

Las acciones de asignación se clasifican según sea el tipo de expresiones en: aritméticas, lógicas y de caracteres.

Asignación aritmética

Las expresiones en las operaciones de asignación son aritméticas:

AMN ← 3 + 14 + 8 se evalúa la expresión 3 + 14 + 8 y se asigna a la variable AMN, es decir,

25 será el valor que toma AMN

TER1 ← 14.5 + 8

TER2 ← 0.75 * 3.4

COCIENTE ← TER1/TER2

Se evalúan las expresiones 14.5 + 8 y 0.75 * 3.4 y en la tercera acción se dividen los resultados de cada expresión y se asigna a la variable COCIENTE, es decir, las tres operaciones equivalen a COCIENTE ← (14.5 + 8)/ (0.75 * 3.4).

Otro ejemplo donde se pueden comprender las modificaciones de los valores almacenados en una variable es el siguiente:

A ← 0 la variable A toma el valor 0

N ← 0 la variable N toma el valor 0

A ← N + 1 la variable A toma el valor 0 + 1, es decir 1.

El ejemplo anterior se puede modificar para considerar la misma variable en ambos lados del operador de asignación:

N ← 2
N ← N + 1
En la primera acción N toma el valor 2 y en la segunda se evalúa la expresión N + 1, que tomará el valor 2 + 1 = 3 y se asignará nuevamente a N, que tomará el valor 3.

Asignación lógica

La expresión que se evalúa en la operación de asignación es lógica. Supóngase que M, N y P son variables de tipo lógico.

M ← 8 < 5
N ← M o (7 <= 12)
P ← 7 > 6

Tras evaluar las operaciones anteriores, las variables M, N y P tomarán los valores falso, verdadero, verdadero.

Asignación de cadenas de caracteres

La expresión que se evalúa es de tipo cadena:

x ← '12 de octubre de 1942'

La acción de asignación anterior asigna la cadena de caracteres '12 de octubre de 1942' a la variable tipo cadena x.

Asignación múltiple

Todos los lenguajes modernos admiten asignaciones múltiples y con combinaciones de operadores, además de la asignación única con el operador ← . Así se puede usar el operador de asignación (←) precedido por cualquiera de los siguientes operadores aritméticos: +, –, *, /, %. La sintaxis es la siguiente:

<nombre_variable> ← <variable> <operador> <expresión>

y es equivalente a:

variable operador ← expresión

Conversión de tipo

En las asignaciones no se pueden asignar valores a una variable de un tipo incompatible al suyo. Se presentará un error si se trata de asignar valores de tipo carácter a una variable numérica o un valor numérico a una variable tipo carácter.

EJEMPLO 3.11

¿Cuáles son los valores de A, B y C *después de la ejecución de las siguientes operaciones?*

```
A ← 3
B ← 4
C ← A + 2 * B
C ← C + B
B ← C - A
A ← B * C
```

En las dos primeras acciones A y B toman los valores 3 y 4.

```
C ← A + 2 * B
```

*la expresión A + 2 * B tomará el valor 3 + 2 * 4 = 3 + 8 = 11*

```
C ← 11
```

La siguiente acción

```
C ← C + B
```

producirá un valor de `11 + 4 = 15`

```
C ← 15
```

En la acción `B ← C - A` se obtiene para `B` el valor `15 - 3 = 12` y por último:

```
A ← B * C
```

`A` tomará el valor `B * C`, es decir, `12 * 15 = 180`; por consiguiente, el último valor que toma `A` será `180`.

EJEMPLO 3.12

¿Cuál es el valor de x después de las siguientes operaciones?

```
x ← 2
x ← cuadrado(x + x)
x ← raiz2(x + raiz2(x) + 5)
```

Los resultados de cada expresión son:

```
x ← 2                          x toma el valor 2
x ← cuadrado(2 + 2)            x toma el valor 4 al cuadrado; es decir 16
x ← raiz2(16 + raiz2(16) + 5)
```

en esta expresión se evalúa primero `raiz2(16)`, que produce 4 y, por último, `raiz2(16+4+5)` proporciona `raiz2(25)`, es decir, 5. Los resultados de las expresiones sucesivas anteriores son:

```
x ← 2
x ← 16
x ← 5
```

3.9 Entrada y salida de información.

Los cálculos que realizan las computadoras requieren para ser útiles la entrada de los datos necesarios para ejecutar las operaciones que posteriormente se convertirán en resultados, es decir, salida.

Las operaciones de entrada permiten leer determinados valores y asignarlos a determinadas variables. Esta entrada se conoce como operación de lectura (read). Los datos de entrada se introducen al procesador mediante dispositivos de entrada (teclado, tarjetas perforadas, unidades de disco, etc.). La salida puede aparecer en un dispositivo de salida (pantalla, impresora, etc.). La operación de salida se denomina escritura (write).

En la escritura de algoritmos las acciones de lectura y escritura se representan por los formatos siguientes:

leer (lista de variables de entrada)

escribir (lista de variables de salida)

Así, por ejemplo:

leer (A, B, C)

representa la lectura de tres valores de entrada que se asignan a las variables A, B y C.

escribir ('hola Vargas')

visualiza en la pantalla —o escribe en el dispositivo de salida— el mensaje 'hola Vargas'.

Nota 1

Si se utilizaran las palabras reservadas en inglés, como suele ocurrir en los lenguajes de programación, se deberá sustituir

leer escribir
por
read write o bien print

Nota 2

Si no se especifica el tipo de dispositivo del cual se leen o escriben datos, los dispositivos de E/S por defecto son el teclado y la pantalla.

3.10 Escritura de algoritmos/programas.

de modo que su lectura facilite considerablemente el entendimiento del algoritmo y su posterior codificación en un lenguaje de programación.

Los algoritmos deben ser escritos en lenguajes similares a los programas. En nuestro libro utilizaremos esencialmente el lenguaje algorítmico, basado en pseudocódigo, y la estructura del algoritmo requerirá la lógica de los programas escritos en el lenguaje de programación estructurado; por ejemplo, Pascal.

Un algoritmo constará de dos componentes: una cabecera de programa y un bloque algoritmo. La cabecera de programa es una acción simple que comienza con la palabra algoritmo. Esta palabra estará seguida por el nombre asignado al programa completo. El

bloque algoritmo es el resto del programa y consta de dos componentes o secciones: las acciones de declaración y las acciones ejecutables.

Las declaraciones definen o declaran las variables y constantes que tengan nombres. Las acciones ejecutables son las acciones que posteriormente deberá realizar la computación cuando el algoritmo convertido en programa se ejecute.

algoritmo
cabecera del programa
sección de declaración
sección de acciones

3.11 Cabecera del programa.

Todos los algoritmos y programas deben comenzar con una cabecera en la que se exprese el identificador o nombre correspondiente con la palabra reservada que señale el lenguaje. En los lenguajes de programación, la palabra reservada suele ser program. En Algorítmica se denomina algoritmo.

algoritmo DEMO1

3.12 Declaración de variables.

En esta sección se declaran o describen todas las variables utilizadas en el algoritmo, listándose sus nombres y especificando sus tipos. Esta sección comienza con la palabra reservada var (abreviatura de variable) y tiene el formato

var
tipo-1 : lista de variables-1
tipo-2 : lista de variables-2
.
.
tipo-n : lista de variables-n

donde cada lista de variables es una variable simple o una lista de variables separadas por comas y cada tipo es uno de los tipos de datos básicos (entero, real, char o boolean). Por ejemplo, la sección de declaración de variables

```
var
entera : Numero_Empleado
real : Horas
real : Impuesto
real : Salario
```

o de modo equivalente

```
var
entera : Numero_Empleado
real : Horas, Impuesto, Salario
```

declara que sólo las tres variables Hora, Impuesto y Salario son de tipo real.

Es una buena práctica de programación utilizar nombres de variables significativos que sugieran lo que ellas representan, ya que eso hará más fácil y legible el programa.

También es buena práctica incluir breves comentarios que indiquen cómo se utiliza la variable.

```
var
entera : Numero_Empleado // número de empleado
real : Horas, // horas trabajadas
Impuesto, // impuesto a pagar
Salario // cantidad ganada
```

3.13 Declaración de constantes numéricas.

En esta sección se declaran todas las constantes que tengan nombre. Su formato es

```
const
pi = 3.141592
```

tamaño = 43

horas = 6.50

Los valores de estas constantes ya no pueden variar en el transcurso del algoritmo.

3.14 Declaración de constantes y variables carácter.

Las constantes de carácter simple y cadenas de caracteres pueden ser declaradas en la sección del programa const, al igual que las constantes numéricas.

```
const
        estrella = '*'
        frase = '12 de octubre'
        mensaje = 'Hola mi nene'
```

Las variables de caracteres se declaran de dos modos:

1. Almacenar un solo carácter.

var carácter : nombre, inicial, nota, letra

Se declaran nombre, inicial, nota y letra, que almacenarán sólo un carácter.

2. Almacenar múltiples caracteres (cadenas). El almacenamiento de caracteres múltiples dependerá del lenguaje de programación. Así, en los lenguajes

VB 6.0/VB .NET (VB, Visual Basic)

```
Dim var1 As String

Var1 = "Pepe Luis García Rodriguez"
```

Pascal formato tipo array o arreglo (véase Capítulo 8).

Existen algunas versiones de Pascal, como es el caso de Turbo Pascal, que tienen implementados un tipo de datos denominados string (cadena) que permite declarar variables de caracteres o de cadena que almacenan palabras compuestas de diferentes caracteres.

var nombre : string[20]; en Turbo Pascal

var cadena : nombre[20]; en pseudocódigo

3.15 Comentarios.

La documentación de un programa es el conjunto de información interna externa al programa, que facilitará su posterior mantenimiento y puesta a punto. La documentación puede ser interna y externa.

La documentación externa es aquella que se realiza externamente al programa y con fines de mantenimiento y actualización; es muy importante en las fases posteriores a la puesta en marcha inicial de un programa. La documentación interna es la que se acompaña en el código o programa fuente y se realiza a base de comentarios significativos.

Estos comentarios se representan con diferentes notaciones, según el tipo de lenguaje de programación.

Visual Basic 6 / VB .NET

1. Los comentarios utilizan un apóstrofe simple y el compilador ignora todo lo que viene después de ese carácter

'Este es un comentario de una sola línea

Dim Mes As String 'comentario después de una línea de código

.....

2. También se admite por guardar compatibilidad con versiones antiguas de BASIC y Visual Basic la palabra reservada Rem

Rem esto es un comentario

C/C++ y C#

Existen dos formatos de comentarios en los lenguajes C y C++:

1. Comentarios de una línea (comienzan con el carácter //)

// Programa 5.0 realizado por el Señor Mackoy

// en Carchelejo (Jaén)en las Fiestas de Agosto

// de Moros y Cristiano

2. Comentarios multilínea (comienzan con los caracteres /* y terminan con los caracteres */, todo lo encerrado entre ambos juegos de caracteres son comentarios)

/* El maestro Mackoy estudió el Bachiller en el mismo Instituto donde dio clase
Don Antonio Machado, el poeta */

Java

1. Comentarios de una línea

// comentarios sobre la Ley de Protección de Datos

2. Comentarios multilíneas

/* El pueblo de Mr. Mackoy está en Sierra Mágina, y produce uno
de los mejores aceites de oliva del mundo mundial */

3. Documentación de clases

/**

Documentación de la clase

*/

Pascal

Los comentarios se encierran entre los símbolos

(* *)

o bien

{ }
(* autor J.R. Mackoy *)
{subrutina ordenacion}

Modula-2

Los comentarios se encierran entre los símbolos

(* *)

UNIDAD IV ESTRUCTURAS DE CONTROL DE UN PROGRAMA

Muchos avances han ocurrido en los fundamentos teóricos de programación desde la aparición de los lenguajes de alto nivel a finales de la década de los cincuenta. Uno de los más importantes avances fue el reconocimiento a finales de los sesenta de que cualquier algoritmo, no importaba su complejidad, podía ser construido utilizando combinaciones de tres estructuras de control de flujo estandarizadas (secuencial, selección, repetitiva o iterativa) y una cuarta denominada, invocación o salto ("jump"). Las sentencias de selección son: si (if) y según-sea (switch); las sentencias de repetición o iterativas son: desde (for), mientras (while), hacer-mientras (do-while) o repetir-hasta que (repeat-until); las sentencias de salto o bifurcación incluyen romper (break), continuar (continue), ir-a (goto), volver (return) y lanzar (throw).

El término flujo de control se refiere al orden en que se ejecutan las sentencias del programa. Otros términos utilizados son secuenciación y control del flujo. A menos que se especifique expresamente, el flujo normal de control de todos los programas es el secuencial. Este término significa que las sentencias se ejecutan en secuencia, una después de otra, en el orden en que se sitúan dentro del programa. Las estructuras de selección, repetición e invocación permiten que el flujo secuencial del programa sea modificado en un modo preciso y definido con anterioridad. Como se puede deducir fácilmente, las estructuras de selección se utilizan para seleccionar cuáles sentencias se han de ejecutar a continuación y las estructuras de repetición (repetitivas o iterativas) se utilizan para repetir un conjunto de sentencias.

Hasta este momento, todas las sentencias se ejecutaban secuencialmente en el orden en que estaban escritas en el código fuente. Esta ejecución, como ya se ha comentado, se denomina ejecución secuencial. Un programa basado en ejecución secuencial, siempre ejecutará exactamente las mismas acciones; es incapaz de reaccionar en respuesta a condiciones actuales. Sin embargo, la vida real no es tan simple. Normalmente, los

programas necesitan alterar o modificar el flujo de control en un programa. Así, en la solución de muchos problemas se deben tomar acciones diferentes dependiendo del valor de los datos. Ejemplos de situaciones simples son: cálculo de una superficie sólo si las medidas de los lados son positivas; la ejecución de una división se realiza, sólo si el divisor no es cero; la visualización de mensajes diferentes depende del valor de una nota recibida, etc.

Una bifurcación ("branch", en inglés) es un segmento de programa construida con una sentencia o un grupo de sentencias. Una sentencia de bifurcación se utiliza para ejecutar una sentencia de entre varias o bien bloques de sentencias. La elección se realiza dependiendo de una condición dada. Las sentencias de bifurcación se llaman también sentencias de selección o sentencias de alternación o alternativas.

4.1 Estructura secuencial.

Una estructura secuencial es aquella en la que una acción (instrucción) sigue a otra en secuencia. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente y así sucesivamente hasta el final del proceso. La estructura secuencial tiene una entrada y una salida. Su representación gráfica se muestra en las Figuras 4.1, 4.2 y 4.3.

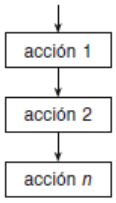


Figura 4.1. Estructura secuencial.

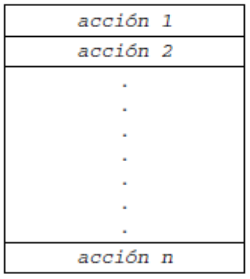


Figura 4.2. Diagrama N-S de una estructura secuencial.

```
inicio
  <acción 1>
  <acción 2>
fin
```

Figura 4.3. Pseudocódigo de una estructura secuencial.

EJEMPLO 4.1

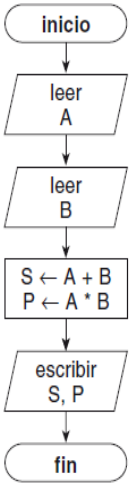
Cálculo de la suma y producto de dos números.

La suma `S` de dos números es `S = A+B` y el producto `P` es `P = A*B`. El pseudocódigo y el diagrama de flujo correspondientes se muestran a continuación:

Pseudocódigo

```
inicio
  leer(A)
  leer(B)
  S ← A + B
  P ← A * B
  escribir(S, P)
fin
```

Diagrama de flujo



4.2 Estructuras selectivas.

La especificación formal de algoritmos tiene realmente utilidad cuando el algoritmo requiere una descripción más complicada que una lista sencilla de instrucciones. Este es el caso cuando existen un número de posibles alternativas resultantes de la evaluación de una determinada condición. Las estructuras selectivas se utilizan para tomar decisiones lógicas; de ahí que se suelan denominar también estructuras de decisión o alternativas.

En las estructuras selectivas se evalúa una condición y en función del resultado de la misma se rea liza una opción u otra. Las condiciones se especifican usando expresiones lógicas. La

representación de una estructura selectiva se hace con palabras en pseudocódigo (if, then, else o bien en español si, entonces, si_no), con una figura geométrica en forma de rombo o bien con un triángulo en el interior de una caja rectangular. Las estructuras selectivas o alternativas pueden ser:

• simples,
• dobles,
• múltiples.

La estructura simple es si (if) con dos formatos: Formato Pascal, si-entonces (if-then) y formato C, si (if). La estructura selectiva doble es igual que la estructura simple si a la cual se le añade la cláusula si-no (else). La estructura selectiva múltiple es según_sea (switch en lenguaje C, case en Pascal).

4.3 Alternativa simple (si-entonces/if-then).

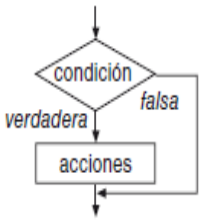
La estructura alternativa simple si-entonces (en inglés if-then) ejecuta una determinada acción cuando se cumple una determinada condición. La selección si-entonces evalúa la condición y

• si la condición es verdadera, entonces ejecuta la acción S1 (o acciones caso de ser S1 una acción compuesta y constar de varias acciones),

• si la condición es falsa, entonces no hacer nada.

Las representaciones gráficas de la estructura condicional simple se muestran en la Figura 4.4.

a)



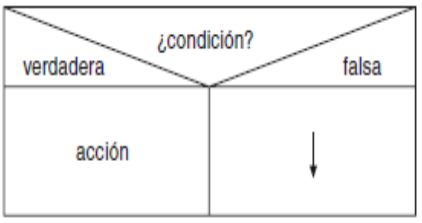
b) *Pseudocódigo en español*

```
si <condición> entonces
   <acción S1>
fin_si
```

Pseudocódigo en inglés

```
if <condición> then
   <acción S1>
endif
```

c)



Pseudocódigo en español

```
//S1 accion compuesta
si <condición> entonces
   <acción S11>
   <acción S12>
      .
      .
      .
   <acción S1n>
fin_si
```

Figura 4.4. Estructuras alternativas simples: *a*) Diagrama de flujo; *b*) Pseudocódigo; *c*) Diagrama N-S.

Obsérvese que las palabras del pseudocódigo si y fin_si se alinean verticalmente indentando (sangrando) la <acción> o bloque de acciones.

Diagrama de sintaxis

Sentencia **if_simple::=**

```
1. si (<expresión_lógica>)
   inicio
      <sentencia>
   fin

   Sentencia_compuesta ::=
                    inicio
                         <Sentencias>
                    fin
```

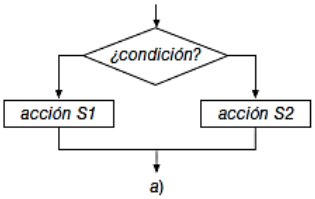
```
2. si <expresión lógica> entonces
      <Sentencia_compuesta>
   fin-si
```

Sintaxis en lenguajes de programación

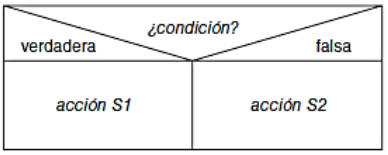
Pseudocódigo	*Pascal*	*C/C++*
si *(condición)* **entonces**	**if** *(condición)* **then**	**if** *(condición)*
	begin	{
acciones	*sentencias*	*sentencias*
fin-si	**end**	}

4.4 Alternativa doble (si-entonces-sino/if-then-else).

La estructura anterior es muy limitada y normalmente se necesitará una estructura que permita elegir entre dos opciones o alternativas posibles, en función del cumplimiento o no de una determinada condición. Si la condición C es verdadera, se ejecuta la acción S1 y, si es falsa, se ejecuta la acción S2 (véase Figura 4.5).



a)



c)

Pseudocódigo en español

```
si <condicion> entonces
   <accion S1>
si _ no
   <accion S2>
fin _ si
```

Pseudocódigo en inglés

```
if <condicion> then
   <accion S1>
else
   <accion S2>
endif
```

Pseudocódigo en español

```
//S1 accion compuesta
si <condicion> entonces
   <accion S11>
   <accion S12>
   .
   .
   .
   <acción S1n>
si _ no
   <accion S21>
   <accion S22>
   .
   .
   .
   <acción S2n>
fin _ si
```

b)

Figura 4.5. Estructura alternativa doble: *a*) diagrama de flujo; *b*) pseudocódigo; *c*) diagrama N-S.

Obsérvese que en el pseudocódigo las acciones que dependen de entonces y si_no están indentadas en relación con las palabras si y fin_si; este procedimiento aumenta la legibilidad de la estructura y es el medio más idóneo para representar algoritmos.

EJEMPLO 4.3

Resolución de una ecuación de primer grado.

Si la ecuación es $ax + b = 0$, a y b son los datos, y las posibles soluciones son:

```
• a <> 0                x = -b/a
• a = 0 b <> 0          entonces "solución imposible"
• a = 0 b = 0           entonces "solución indeterminada"
```

El algoritmo correspondiente será

```
algoritmo RESOL1
var
  real : a, b, x
inicio
  leer (a, b)
  si a <> 0 entonces
     x ← -b/a
     escribir(x)
  si_no
     si b <> 0 entonces
         escribir ('solución imposible')
     si_no
         escribir ('solución indeterminada')
     fin_si
  fin_si
fin
```

4.5 Alternativa múltiple (según_sea, caso de/case).

Con frecuencia —en la práctica— es necesario que existan más de dos elecciones posibles (por ejemplo, en la resolución de la ecuación de segundo grado existen tres posibles alternativas o caminos a seguir, según que el discriminante sea negativo, nulo o positivo). Este problema, como se verá más adelante, se podría resolver por estructuras alternativas simples o dobles, anidadas o en cascada; sin embargo, este método si el número de alternativas es grande puede plantear serios problemas de escritura del algoritmo y naturalmente de legibilidad.

La estructura de decisión múltiple evaluará una expresión que podrá tomar n valores distintos, 1, 2, 3, 4, ..., n. Según que elija uno de estos valores en la condición, se realizará

una de las n acciones, o lo que es igual, el flujo del algoritmo seguirá un determinado camino entre los n posibles.

Los diferentes modelos de pseudocódigo de la estructura de decisión múltiple se representan en las Figuras 4.6 y 4.7.

***Sentencia* switch** (C , C++, Java, C#)

```
switch (expresión)
{
  case valor1:
    sentencia1;
    sentencia2;
    sentencia3;
```

Modelo 1:

```
según_sea expresion (E) hacer
     e1: accion S11
         accion S12
         .
         .
         accion S1a
     e2: accion S21
         accion S22
         .
         .
         accion S2b
     .
     .
     en: accion S31
         accion S32
         .
         .
         accion S3p
     si-no
         accion Sx
fin_según
```

Modelo 2 (simplificado):

```
según E hacer
       .
       .
       .
fin_según
```

Modelo 3 (simplificado):

```
opción E de
 .
 .
fin_opción
```

Modelo 4 (simplificado):

```
caso_de E hacer
 .
 .
 .
fin_caso
```

Modelo 5 (simplificado):

```
si E es n hacer
 .
 .
 .
fin_si
```

Figura 4.6. Estructuras de decisión múltiple.

Modelo 6:

```
según_sea (expresión) hacer
   caso expresión constante :
      [Sentencia
       sentencia
       ...
       sentencia de ruptura | sentencia ir_a ]
   caso expresión constante :
      [Sentencia
       sentencia
       ...
       sentencia de ruptura | sentencia ir_a ]
   caso expresión constante :
      [Sentencia
       ...
       sentencia
       sentencia de ruptura | sentencia ir_a ]
   [otros:
      [Sentencia
       ...
       sentencia
       sentencia de ruptura | sentencia ir_a ]
fin_según
```

Diagrama de flujo

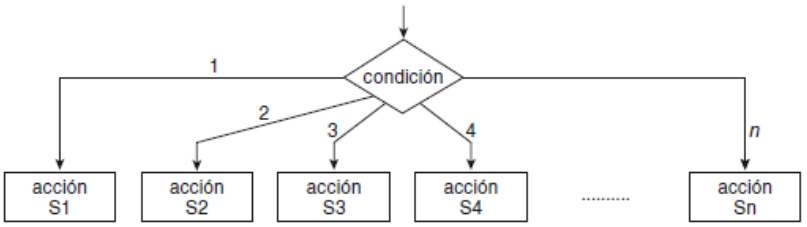
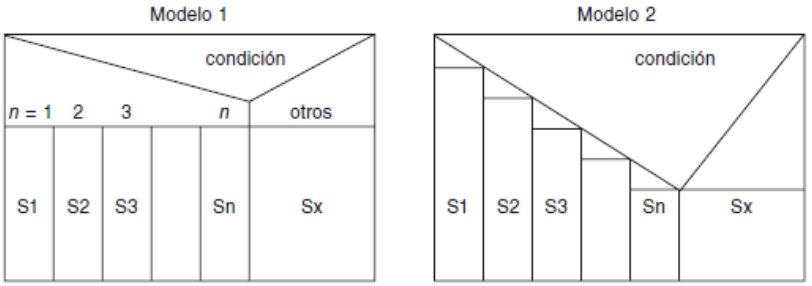


Diagrama N-S



Pseudocódigo

En inglés la estructura de decisión múltiple se representa:

```
   case expresión of
     [e1]: acción S1

   [e2]: acción S2
   .
   .
   [en]: acción Sn
   otherwise
       acción Sx
end_case
```

```
case expresión of
  [e1]: acción S1

  [e2]: acción S2

  [en]: acción Sn
  else
      acción Sx
end_case
```

EJEMPLO 4.9

Se desea diseñar un algoritmo que escriba los nombres de los días de la semana en función del valor de una variable `DIA` *introducida por teclado.*

Los días de la semana son 7; por consiguiente, el rango de valores de `DIA` será `1 .. 7`, y caso de que `DIA` tome un valor fuera de este rango se deberá producir un mensaje de error advirtiendo la situación anómala.

```
algoritmo DiasSemana
var
  entero: DIA
inicio
  leer(DIA)
  según_sea DIA hacer
  1: escribir('LUNES')
  2: escribir('MARTES')
  3: escribir('MIERCOLES')
  4: escribir('JUEVES')
  5: escribir('VIERNES')
  6: escribir('SABADO')
  7: escribir('DOMINGO')
  sí-no
      escribir('ERROR')
  fin_según
fin
```

EJEMPLO 4.10

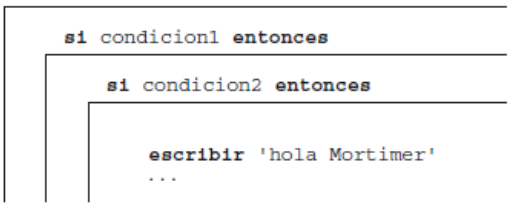
Se desea convertir las calificaciones alfabéticas A, B, C, D, E y F a calificaciones numéricas 4, 5, 6, 7, 8 y 9 respectivamente.

Los valores de `A`, `B`, `C`, `D`, `E` y `F` se representarán por la variable `LETRA`, el algoritmo de resolución del problema es:

```
algoritmo Calificaciones
var
  carácter: LETRA
  entero: calificación
inicio
  leer(LETRA)
  según_sea LETRA hacer
    'A': calificación ← 4
    'B': calificación ← 5
    'C': calificación ← 6
    'D': calificación ← 7
    'E': calificación ← 8
    'F': calificación ← 9
```

4.6 Estructuras de decisión anidadas (en escalera).

Las estructuras de selección si-entonces y si-entonces-si_no implican la selección de una de dos alternativas. Es posible también utilizar la instrucción si para diseñar estructuras de selección que contengan más de dos alternativas. Por ejemplo, una estructura si-entonces puede contener otra estructura si-entonces, y esta estructura si-entonces puede contener otra, y así sucesivamente cualquier número de veces; a su vez, dentro de cada estructura pueden existir diferentes acciones.



Las estructuras si interiores a otras estructuras si se denominan anidadas o encajadas:

```
si <condicion1> entonces
        si <condicion2> entonces
        .
        .
        .
                <acciones>
        fin_si
fin_si
```

Una estructura de selección de n alternativas o de decisión múltiple puede ser construida utilizando una estructura si con este formato:

```
si <condicion1> entonces
        <acciones>
si_no
        si <condicion2> entonces
                <acciones>
        si_no
                si <condicion3> entonces
                        <acciones>
                si_no
                        .
                        .
                        .
                fin_si
        fin_si
fin_si
```

Una estructura selectiva múltiple constará de una serie de estructuras si, unas interiores a otras. Como las estructuras si pueden volverse bastante complejas para que el algoritmo sea claro, será preciso utilizar indentación (sangría o sangrado), de modo que exista una correspondencia entre las palabras reservadas si y fin_si, por un lado, y entonces y si_no, por otro.

La escritura de las estructuras puede variar de unos lenguajes a otros, por ejemplo, una estructura si admite también los siguientes formatos:

```
si <expresion booleana1> entonces
  <acciones>
si_no
  si <expresion booleana2> entonces
    <acciones>
  si_no
    si <expresion booleana3> entonces
      <acciones>
    si_no
      <acciones>
    fin_si
  fin_si
fin_si
```

o bien

```
si <expresion booleana1> entonces
  <acciones>

si_no si <expresion booleana2> entonces
  <acciones>
  fin_si
.
.
.
fin_si
```

EJEMPLO 4.15

Diseñar un algoritmo que lea tres números A, B, C *y visualice en pantalla el valor del más grande. Se supone que los tres valores son diferentes.*

Los tres números son A, B y C; para calcular el más grande se realizarán comparaciones sucesivas por parejas.

```
algoritmo Mayor
var
  real: A, B, C, Mayor
inicio
  leer(A, B, C)
  si A > B entonces
    si A > C entonces
      Mayor ← A          //A > B, A > C
    si_no
      Mayor ← C          //C >= A > B
    fin_si
  si_no
    si B > C entonces
      Mayor ← B          //B >= A, B > C
    si_no
      Mayor ← C          //C >= B >= A
    fin_si
  fin_si
  escribir('Mayor:', Mayor)
fin
```

4.7 Estructura mientras ("while").

La estructura repetitiva mientras (en inglés while o dowhile: hacer mientras) es aquella en que el cuerpo del bucle se repite mientras se cumple una determinada condición. Cuando se ejecuta la instrucción mientras, la primera cosa que sucede es que se evalúa la condición (una expresión booleana). Si se evalúa falsa, no se toma ninguna acción y el programa prosigue en la siguiente instrucción del bucle. Si la expresión booleana es verdadera, entonces se ejecuta el cuerpo del bucle, después de lo cual se evalúa de nuevo la expresión booleana. Este proceso se repite una y otra vez mientras la expresión booleana (condición) sea verdadera. El ejemplo anterior quedaría así ysus representaciones gráficas como las mostradas en la Figura 5.1.

EJEMPLO 5.1

Leer por teclado un número que represente una cantidad de números que a su vez se leerán también por teclado. Calcular la suma de todos esos números.

```
algoritmo suma_numeros
var
  entero : N, TOTAL
  real : numero, SUMA
inicio
  leer(N)
  {leer numero total N}
  TOTAL ← N
  SUMA ← 0

  mientras TOTAL > 0 hacer
    leer(numero)
    SUMA ← SUMA + numero
    TOTAL ← TOTAL - 1
  fin_mientras
  escribir('La suma de los', N, 'numeros es', SUMA)
fin
```

En el caso anterior, como la variable TOTAL se va decrementando y su valor inicial era N, cuando tome el valor 0, significará que se han realizado N iteraciones, o, lo que es igual, se han sumado N números y el bucle se debe parar o terminar.

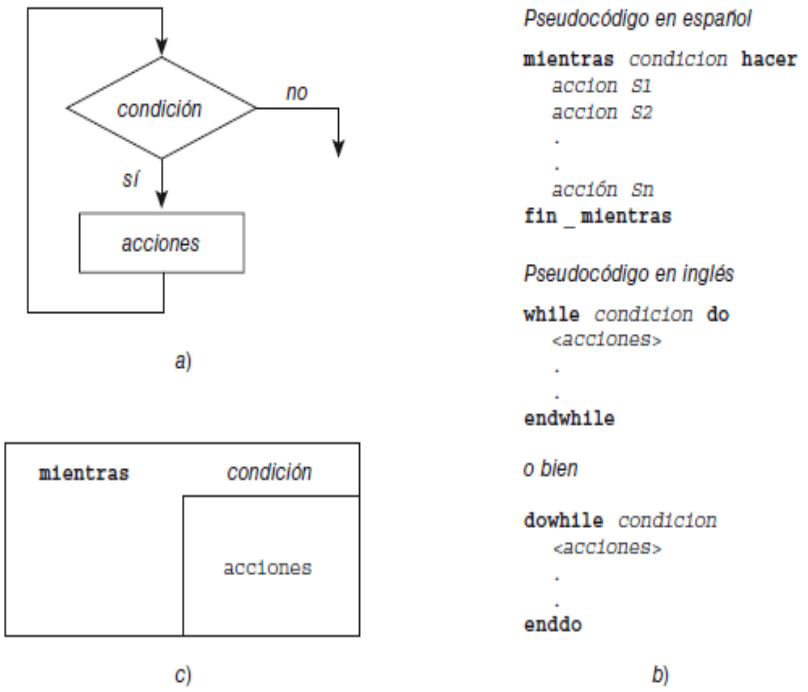


Figura 5.1. Estructura **mientras**: *a*) diagrama de flujo; *b*) pseudocódigo; *c*) diagrama N-S.

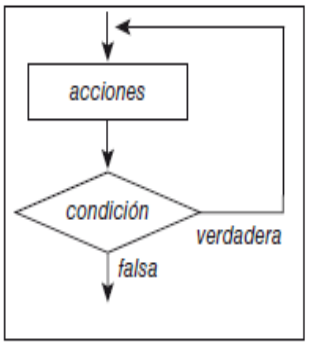
4.8 Estructura hacer-mientras ("do-while").

El bucle mientras al igual que el bucle desde que se verá con posterioridad evalúan la expresión al comienzo del bucle de repetición; siempre se utilizan para crear bucle pre-test. Los bucles pre-test se denominan también bucles controlados por la entrada. En numerosas ocasiones se necesita que el conjunto de sentencias que componen el cuerpo del bucle se ejecuten al menos una vez sea cual sea el valor de la expresión o condición de evaluación. Estos bucles se denominan bucles post-test o bucles controlados por la salida. Un caso típico es el bucle hacer-mientras (do-while) existente en lenguajes como C/C++, Java o C#.

El bucle hacer-mientras es análogo al bucle mientras y el cuerpo del bucle se ejecuta una y otra vez mientras la condición (expresión booleana) sea verdadera. Existe, sin embargo, una gran diferencia y es que el cuerpo del bucle está encerrado entre las palabras reservadas hacer y mientras, de modo que las sentencias de dicho cuerpo se ejecutan, al menos una vez, antes de que se evalúe la expresión booleana. En otras palabras, el cuerpo del bucle siempre se ejecuta, al menos una vez, incluso aunque la expresión booleana sea falsa.

Regla

El bucle hacer-mientras se termina de ejecutar cuando el valor de la condición es falsa. La elección entre un bucle mientras y un bucle hacer-mientras depende del problema de cómputo a resolver. En la mayoría de los casos, la condición de entrada del bucle mientras es la elección correcta. Por ejemplo, si el bucle se utiliza para recorrer una lista de números (o una lista de cualquier tipo de objetos), la lista puede estar vacía, en cuyo caso las sentencias del bucle nunca se ejecutarán. Si se aplica un bucle hacer-mientras nos conduce a un código de errores.



a) Diagrama de flujo de una sentencia **hacer-mientras**

```
hacer
   <acciones>
mientras  (<expresión>)
```

b) Pseudocódigo de una sentencia **hacer-mientras**

Figura 5.2. Estructura **hacer-mientras**: *a*) diagrama de flujo; *b*) pseudocódigo.

Al igual que en el caso del bucle mientras la sentencia en el interior del bucle puede ser simple o compuesta. Todas las sentencias en el interior del bucle se ejecutan al menos una vez antes de que la expresión o condición se evalúe. Entonces, si la expresión es verdadera (un valor distinto de cero, en C/C++) las sentencias del cuerpo del bucle se ejecutan una vez más. El proceso continúa hasta que la expresión evaluada toma el valor falso (valor cero en C/C++). El diagrama de control del flujo se ilustra en la Figura 5.2, donde se muestra el funcionamiento de la sentencia hacer-mientras. La Figura 5.3 representa un diagrama de sintaxis con notación BNF de la sentencia hacer-mientras.

Sentencia hacer-mientras::=
hacer
<cuerpo del bucle>
mientras *(<condición_del_bucle>)*

donde

<cuerpo del bucle>::= <sentencia>
::= <sentencia_compuesta>

<condición del bucle>::= <expresión booleana>

Nota: el cuerpo del bucle se repite mientras <condición del bucle> sea verdadero.

Figura 5.3. Diagrama de sintaxis de la sentencia **hacer-mientras**.

4.9 Estructura repetir ("repeat").

condición de repetición. En la estructura mientras si el valor de la expresión booleana es inicialmente falso, el cuerpo del bucle no se ejecutará; por ello, se necesitan otros tipos de estructuras repetitivas.

La estructura repetir (repeat) se ejecuta hasta que se cumpla una condición determinada que se comprueba al final del bucle (Figura 5.4).

El bucle repetir-hasta_que se repite mientras el valor de la expresión booleana de la condición sea falsa, justo la opuesta de la sentencia mientras.

```
algoritmo repetir
var
        real : numero
        entero: contador
inicio
        contador ← 1
        repetir
                leer(numero)
                contador ← contador + 1
        hasta_que contador > 30
        escribir('Numeros leidos 30')
fin
```

En el ejemplo anterior el bucle se repite hasta que el valor de la variable contador exceda a 30, lo que sucederá después de 30 ejecuciones del cuerpo del bucle.

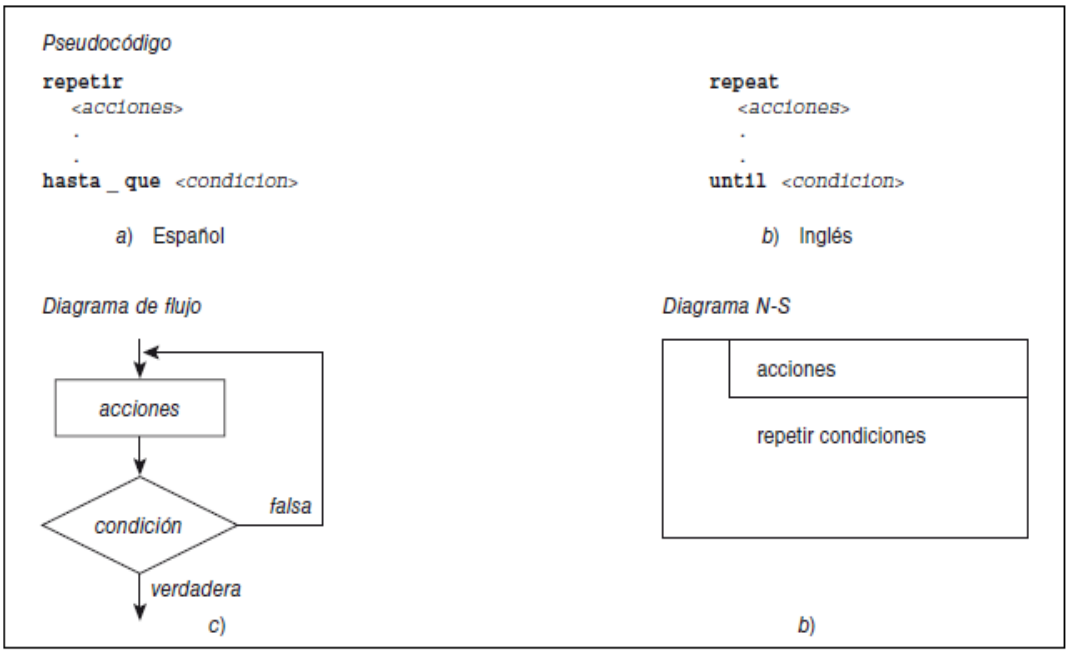


Figura 5.4. Estructura **repetir**: pseudocódigo, diagrama de flujo, diagrama N-S.

EJEMPLO 5.8

Desarrollar el algoritmo necesario para calcular el factorial de un número N que responda a la fórmula:

$$N! = N * (N - 1) * (N - 2), ..., 3 * 2 * 1$$

El algoritmo correspondiente es:

```
algoritmo factorial
var
  entero : I, N
  real : Factorial
inicio
  leer(N)          // N > = 1
  Factorial ← 1
  I ← 1
  repetir
      Factorial ← Factorial * I
      I ← I + 1
  hasta_que I = N + 1
  escribir('El factorial del numero', N, 'es', Factorial)
fin
```

Con una estructura repetir el cuerpo del bucle se ejecuta siempre al menos una vez. Cuando una instrucción repetir se ejecuta, lo primero que sucede es la ejecución del bucle y, a continuación, se evalúa la expresión booleana resultante de la condición. Si se evalúa como falsa, el cuerpo del bucle se repite y la expresión booleana se evalúa una vez. Después

de cada iteración del cuerpo del bucle, la expresión booleana se evalúa; si es verdadera, el bucle termina y el programa sigue en la siguiente instrucción a hasta_que.

Diferencias de las estructuras mientras y repetir

- La estructura `mientras` termina cuando la condición es falsa, mientras que `repetir` termina cuando la condición es verdadera.
- En la estructura `repetir` el cuerpo del bucle se ejecuta siempre al menos una vez; por el contrario, `mientras` es más general y permite la posibilidad de que el bucle pueda no ser ejecutado. Para usar la estructura `repetir` debe estar seguro de que el cuerpo del bucle —bajo cualquier circunstancia— se repetirá al menos una vez.

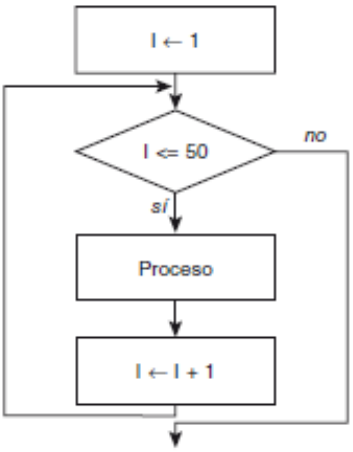
4.10 Estructura desde/para ("for").

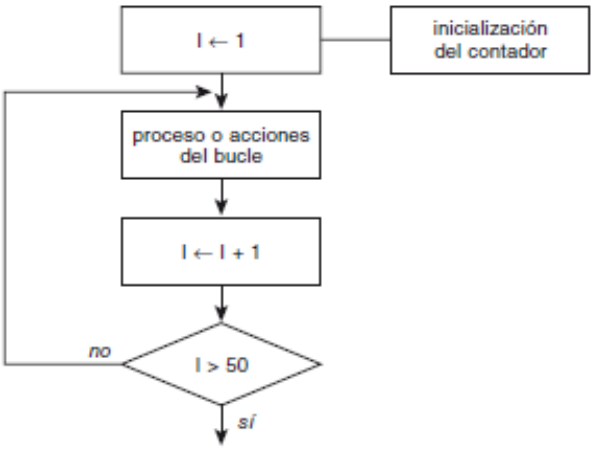
En muchas ocasiones se conoce de antemano el número de veces que se desean ejecutar las acciones de un bucle. En estos casos, en el que el número de iteraciones es fijo, se debe usar la estructura desde o para (for, en inglés).

La estructura desde ejecuta las acciones del cuerpo del bucle un número especificado de veces y de modo automático controla el número de iteraciones o pasos a través del cuerpo del bucle. Las herramientas de programación de la estructura desde o para se muestran en la página siguiente junto a la Figura 5.5.

Otras representaciones de estructuras repetitivas desde/para (for)

Un bucle desde (for) se representa con los símbolos de proceso y de decisión mediante un contador. Así, por ejemplo, en el caso de un bucle de lectura de cincuenta números para tratar de calcular su suma:


I ← 1
I <= 50
no
sí
Proceso
I ← I + 1


I ← 1
inicialización del contador
proceso o acciones del bucle
I ← I + 1
no
I > 50
sí

Pseudocódigo estructura **desde**

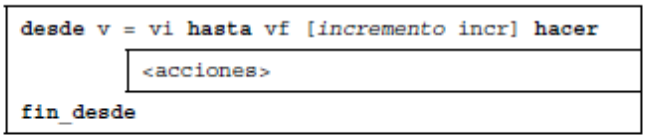
```
desde v ← vi hasta vf [incremento incr] hacer
  <acciones>
    .
    .
    .
fin_desde
v: variable indice
vi, vf: valores inicial y final de la variable
```

a) Modelo 1

```
para v ← vi hasta vf [incremento incr] hacer
  <acciones>
    .
    .
    .
fin_para
```

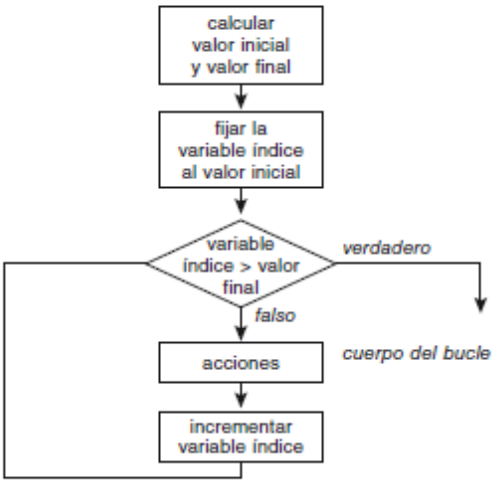
b) Modelo 2

Diagrama N-S, estructura **desde**



b) Modelo 3

Diagrama de flujo, estructura, **desde**



c) Modelo 4

Figura 5.5. Estructura **desde (for)**: *a*) pseudocódigo, *b*) diagrama N-S, *c*) diagrama de flujo.

La estructura desde comienza con un valor inicial de la variable índice y las acciones especificadas se ejecutan, a menos que el valor inicial sea mayor que el valor final. La variable índice se incrementa en uno y si este nuevo valor no excede al final, se ejecutan de nuevo las acciones. Por consiguiente, las acciones específicas en el bucle se ejecutan para cada

valor de la variable índice desde el valor inicial hasta el valor final con el incremento de uno en uno.

El incremento de la variable índice siempre es 1 si no se indica expresamente lo contrario. Dependiendo del tipo de lenguaje, es posible que el incremento sea distinto de uno, positivo o negativo. Así, por ejemplo, FORTRAN admite diferentes valores positivos o negativos del incremento, y Pascal sólo admite incrementos cuyo tamaño es la unidad: bien positivos, bien negativos. La variable índice o de control normalmente será de tipo entero y es normal emplear como nombres las letras I, J, K. El formato de la estructura desde varía si se desea un incremento distinto a 1, bien positivo, bien negativo (decremento).

```
desde v ← vi hasta vf          inc paso hacer      {inc, incremento}
                               dec                 {dec, decremento}
  <acciones>
    .
    .
    .
fin_desde
```

Si el valor inicial de la variable índice es menor que el valor final, los incrementos deben ser positivos, ya que en caso contrario la secuencia de acciones no se ejecutaría. De igual modo, si el valor inicial es mayor que el valor final, el incremento debe ser en este caso negativo, es decir, decremento. Al incremento se le suele denominar también paso ("step", en inglés). Es decir,

desde i ← 20 hasta 10 hacer
<acciones>
fin_desde

no se ejecutaría, ya que el valor inicial es 20 y el valor final 10, y como se supone un incremento positivo, de valor 1, se produciría un error. El pseudocódigo correcto debería ser

desde i ← 20 hasta 10 decremento 1 hacer
<acciones>
fin_desde

4.11 Sentencias de salto interrumpir (break) y continuar (continue).

Las secciones siguientes examinan las sentencias de salto (jump) que se utilizan para influir en el flujo de ejecución durante la ejecución de una sentencia de bucle.

Sentencia interrumpir (break)

En ocasiones, los programadores desean terminar un bucle en un lugar determinado del cuerpo del bucle en vez de esperar que el bucle termine de modo natural por su entrada o por su salida. Un método de conseguir esta acción —siempre utilizada con precaución y con un control completo del bucle— es mediante la sentencia interrumpir (break) que se suele utilizar en la sentencia según_sea (switch).

La sentencia interrumpir se puede utilizar para terminar una sentencia de iteración y cuando se ejecuta produce que el flujo de control salte fuera a la siguiente sentencia inmediatamente a continuación de la sentencia de iteración. La sentencia interrumpir se puede colocar en el interior del cuerpo del bucle para implementar este efecto.

Sintaxis
interrumpir
sentencia_interrumpir::= interrumpir

EJEMPLO 5.13

```
hacer
    escribir ('Introduzca un número de identificiación')
    leer (numId)
    si (numId < 1000 o numId > 1999)entonces
      escribir ('Número no válido ')
      escribir ('Por favor, introduzca otro número')
    si-no
      interrumpir
    fin_si
mientras (expresión cuyo valor sea siempre verdadero)
```

Regla

La sentencia interrumpir (break) se utiliza frecuentemente junto con una sentencia si (if) actuando como una condición interna del bucle.

Bibliografía

Fundamentos de programación	Luis Joyanes Aguilar	McGraw-Hill
Fundamentos de programación	José Alfredo Jiménez Murillo	Alfaomega
Fundamentos de programación C/C++	Ernesto Peñaloza Romero	Alfaomega

TITULO	LINK	AUTOR
¿Qué es un diagrama de flujo?	https://www.youtube.com/watch?v=Kucgc6NpGwc	YouTube
Estructura general de un programa en C	https://www.youtube.com/watch?v=YmFP8buP9CE	YouTube
Lenguaje C: Estructuras de Control de Flujo (switch , if , while , for)	https://www.youtube.com/watch?v=QgwHXz-Jb6A	YouTube