

INTRODUCCION A LA PROGRAMACION

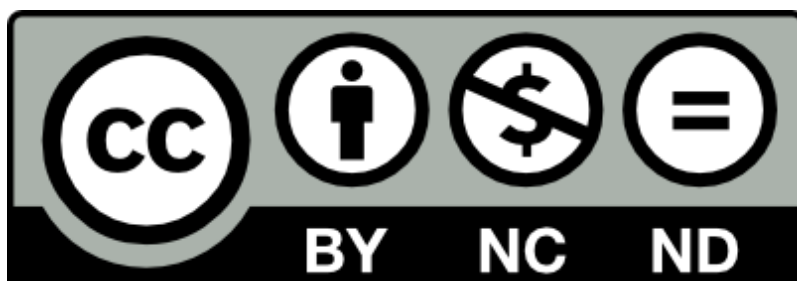
CARLOS E. CIMINO

A blurred office background with a laptop in the foreground. The laptop screen is dark with a small blue icon. The background shows desks, chairs, and other office equipment out of focus.

"No hay que ser un genio para programar"

Introducción a la programación

Carlos E. Cimino



Este documento se encuentra bajo Licencia Creative Commons 4.0 Internacional (CC BY-NC-ND 4.0).

Usted es libre para:

- **Compartir** — copiar y redistribuir el material en cualquier medio o formato.

Bajo los siguientes términos:

- **Atribución** — Usted debe darle crédito a esta obra de manera adecuada, proporcionando un enlace a la licencia, e indicando si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo del licenciante.
- **No Comercial** — Usted no puede hacer uso del material con fines comerciales.
- **Sin Derivar** — Si usted mezcla, transforma o crea nuevo material a partir de esta obra, usted no podrá distribuir el material modificado.

Última modificación: 26 de abril de 2018

Tabla de contenido

Introducción.....	6
Programación.....	7
Definición	7
Un poco de historia.....	7
¿Por dónde empezar?	10
Algoritmos	12
Definición	12
Características	12
Representación.....	13
Preparando el ambiente.....	16
Instalación de PSeInt	16
Configuración de la sintaxis	18
Primer programa	19
Errores.....	21
Flujo secuencial	23
Instrucción de salida	24
Tipos de datos.....	26
Datos numéricos.....	27
Datos alfanuméricos	27
Datos booleanos	29
Convención.....	29
Expresiones aritméticas	30
Operadores aritméticos	30
Expresión vs. Instrucción	31
Variables.....	32
Definición	32
Declaración.....	32
Reglas.....	34
Convención.....	35
Operador de asignación.....	35

Actualizar una variable	38
Instrucción de entrada	39
Operador de concatenación	41
Uso de funciones	41
Funciones para números	43
Funciones para cadenas.....	44
Comentarios.....	45
Integrando los conceptos.....	46
Ejecución paso a paso	47
Generar diagrama de flujo	47
Flujo de selección.....	49
Expresiones booleanas	49
Operadores relacionales.....	50
Flujo de selección simple.....	52
Flujo de selección doble.....	55
Anidamiento de estructuras de selección	58
Operadores lógicos	63
Operador NOT	63
Operador OR.....	64
Operador AND	66
Jerarquía de operadores	69
Flujo de selección múltiple	71
Integrando los conceptos.....	75
Flujo de repetición.....	78
Flujo de repetición Mientras...FinMientras.....	78
Control de ciclos por contador	82
Acumuladores	85
Pruebas de escritorio	88
Control de ciclos por bandera	89
Anidamiento de estructuras de repetición	92
Integrando los conceptos.....	98
Ejemplo integrador final.....	101



Enunciado.....	101
Tips.....	101
Pausar el programa.....	101
Limpiar la pantalla.....	101
El código.....	102
Tabla de códigos.....	105
Tabla de ilustraciones	107

caemci

Introducción

Te doy la bienvenida a este apasionante mundo de la programación. **No hay requisito previo.** Es probable que hayas escuchado que es necesario saber matemáticas o algo por el estilo. Aunque es deseable que tengas un nivel básico de matemática/lógica, quiero llevarte tranquilidad, ya que no necesitás saber ecuaciones diferenciales, derivadas o funciones lineales. No vayas a buscar tus apuntes de la universidad o la secundaria. No creo que tengas la necesidad al principio de saber más que calcular un porcentaje o realizar una regla de tres simple.

Esto es como aprender un instrumento (te lo dice alguien que tiene a la guitarra como hobby). Requiere constancia y paciencia. Al principio parece algo intimidante, pero finalmente es cuestión de tomarle la mano. **Debés aprender a pensar.**

No, no te estoy subestimando. Me refiero a que tenés desarrollar lo que se denomina "**pensamiento computacional**". Esto te ayudará a ser estructurado en tu vida misma, a diseñar soluciones inteligentes y desarrollar un nivel cognitivo que te hará ver las cosas de otra manera.

La programación no es novedosa en absoluto, pero está tomando una trascendencia cada vez mayor: **el lenguaje del futuro es el de las máquinas.** La demanda es brutal, tal es así que cada año quedan miles de puestos vacantes. Pero no tan solo deberías aprender porque la industria requiere de trabajadores. De hecho, podrías armar desde cero tu propio emprendimiento sin mucho requerimiento.

Programar es una herramienta que te da la posibilidad de crear. Una idea que ronde en tu cabeza puede ser plasmada a tu gusto. Te aseguro que se siente muy bien.

Por favor, no te quedes solo con lo que lees en este libro. Es tanto lo que podés hacer y a su vez con posibilidades tan variadas que no existe escrito alguno que pueda mostrarte todas las maneras. Por eso, sé curioso, cambiá algunas cosas, probá otras. Explorá, agregá, modificá, quitá. **¡Jugá!** Vas a aprender más significativamente a través de la prueba y el error que dándote yo todas las pautas.

El cerebro es un músculo, y como tal se ejercita. Un rato cada día, con alguno de descanso en el medio, es suficiente. Es preferible sentarse a programar una hora al día para mantenerse activo.

Por último, ánimo para las chicas. Aquellos mitos que dan cuenta de que los programadores somos en su mayoría hombres se está derribando cada vez más. Aquí cualquier persona de cualquier género es capaz de pensar y desarrollar soluciones a los problemas.

Hecha tal presentación, a mí me dieron ganas de ponerme a programar. **¡Vamos!**

Programación

Definición

La **programación** trata la creación, diseño, codificación, mantenimiento y depuración de un programa o aplicación informática, a través de un **código fuente**.

Un **código fuente** es un conjunto de líneas de texto que describen a la computadora cómo realizar determinada tarea. Se trata de palabras que siguen determinadas reglas sintácticas según el **lenguaje de programación**. Existen multitud de lenguajes de programación de distintas características, que luego detallaré.

El **programa** entonces, que constituye la parte lógica de una computadora (**software**), es ejecutado por la parte física (**hardware**) proveyendo los resultados correspondientes.

Si bien parece que esto se limita solo a las computadoras, deberías saber que existe programación en multitud de dispositivos, como celulares, automóviles, lavarropas, relojes, microondas, etc.

Un poco de historia

En el año 1801, un comerciante textil francés llamado **Joseph Marie Jacquard** inventó un telar gobernado por un sistema de tarjetas perforadas, que presentó en una exhibición industrial de Lyon en 1805. El telar en sí no fue revolucionario, pero sí el sistema de tarjetas perforadas, que permitían el movimiento independiente de los hilos a través de unos ligamentos insertados en diferentes zonas del tejido. Cada tarjeta perforada correspondía a una línea del diseño y la suma de todas las tarjetas creaba el patrón.

Basado en las ideas de Jacquard, entre 1833 y 1842, **Charles Babbage**, matemático y científico británico, impulsa la creación de una máquina capaz de realizar cálculos aritméticos, denominada "**máquina analítica**". La misma tenía dispositivos de entrada basados en las tarjetas perforadas del telar de Jacquard, un procesador aritmético, que calculaba números, una unidad de control que determinaba qué tarea debía ser realizada, un mecanismo de salida y una memoria donde los números podían almacenarse hasta ser procesados. Se la considera como **la primera computadora de la historia**.

Mientras Babbage intentó conseguir financiación para su proyecto, **Lady Ada Lovelace**, matemática hija del lord Byron, se interesó plenamente en él. Ambos matemáticos concebían a la máquina de maneras diferentes: a Babbage no le interesaban mucho sus usos prácticos, pero, por el contrario, Ada se obsesionó con las aplicaciones del invento. Fue la primera en intuir que la máquina significaba un progreso tecnológico. Entendió que podía ser aplicada a todo proceso que implicara

tratar datos, abriendo camino a una nueva ciencia: **la digitalización de la información**. Ada escribió varios programas para la máquina analítica y diferentes historiadores concuerdan que esas instrucciones la convierten en **la primera programadora de computadoras de la historia**.

Un inventor nacido en Estados Unidos, llamado **Herman Hollerith** desarrolló un tabulador electromagnético de tarjetas perforadas que patentó en 1884. Observó que la mayoría de las preguntas en los censos de la época podían contestarse con opciones binarias: **SÍ** o **NO**. Bajo este principio, ideó una tarjeta perforada compuesta por 80 columnas con 2 posiciones, a través de la cual se contestaban este tipo de preguntas.

Los resultados del censo de 1880 en Estados Unidos demandaron unos siete años de análisis y, según proyecciones de aumento de población, se preveía que el censo de 1890 tardaría casi diez años en procesarse. El gobierno estadounidense eligió la máquina tabuladora de Hollerith para elaborar el censo de 1890, logrando que el resultado del recuento y análisis censal de los aproximadamente sesenta millones de habitantes estuviera listo en sólo seis semanas.

En 1896, con el fin de explotar comercialmente su invento, Hollerith fundó la empresa **Tabulating Machine Company**, que tras algunas fusiones, se convierte, en 1924, en la **International Business Machines**, cuya sigla es **IBM**. ¿Te suena?

A partir del siglo XX, más específicamente en la década del 40, se crearon las primeras computadoras electrónicas, basadas en el sistema binario (sistema de numeración de base 2 que trata las distintas combinaciones de VERDADERO y FALSO, o 0 y 1). La **Z3**, a cargo de **Konrad Zuse**, en 1941. La **Atanasoff Berry Computer (ABC)**, obra de **John Vincent Atanasoff** y **Clifford Edward Berry**, entre 1937 y 1942. Luego, llegó el aporte de **Howard Aiken**, quien, en colaboración con IBM, desarrolló la **Mark I** entre 1939 y 1944.

En 1945, el genio matemático **Johann Ludwig Von Neumann** publicó un artículo acerca de una arquitectura que permitía el almacenamiento del programa junto a los datos en un conjunto de celdas que permitía guardar datos binarios, algo que llamó **memoria**. Las computadoras ya no necesitaban recibir las instrucciones una por una, sino que interpretaban los datos binarios guardados en memoria, logrando así, además, una aceleración en los cálculos y prevención de fallas mecánicas.

Ese conjunto de bits (dígitos binarios) se denomina **código máquina** o **lenguaje máquina**. Es lo que finalmente éstas entienden y continúan entendiendo hoy. Para aquella época, la tarea del programador era muy complicada, dado que debía conocer en detalle el hardware de la computadora para poder realizar el programa, paso a paso, en forma de números, lo que hacía el trabajo sumamente propenso a errores y dificultando su comprensión y mantenimiento.

En 1948 se crea en los **Laboratorios Bell** el **transistor**, un componente electrónico semiconductor que impulsó el desarrollo digital, reemplazando a los tubos de vacío.

Para la década del 50, se desarrolla el **lenguaje ensamblador** o **assembler**, que consistía en el uso de **mnemónicos**, es decir, palabras que sustituían los códigos

numéricos de las operaciones a realizar, siendo más fácilmente para los programadores humanos recordar palabras abreviadas como **MOV** (mover) o **INC** (incrementar) que números. El código en lenguaje ensamblador finalmente pasaba por un **ensamblador**, (dada la ambigüedad, el primero es el lenguaje y el segundo el utilitario), que convertía estos mnemónicos en código máquina, directamente interpretable por la computadora. Sin embargo, programar en lenguaje ensamblador seguía siendo engorroso, dado que se continuaba más del lado de la máquina. Además, cada una de ellas era distinta, haciendo que ejecutar un programa en lenguaje ensamblador en una máquina diferente requiriera reescribirlo nuevamente.

En años posteriores comienzan a incorporarse científicos de otras ramas, como la física, la química y las matemáticas, a quienes les resultaba muy complicado enfrentarse a un lenguaje ligado a la computación pura. Nace entonces el concepto de **lenguajes de alto nivel**, que son aquellos que contienen una sintaxis más de lado del ser humano, siendo más fáciles de asimilar.

Para que estos códigos puedan ser ejecutados por la máquina, fue necesario el uso de un **compilador**, es decir, un programa que traducía las instrucciones de alto nivel a lenguaje ensamblador y, posteriormente, a código máquina. La pionera fue nuevamente una mujer, **Grace Hopper**, que desarrolló el primer compilador de la historia en 1952.

Los científicos vieron más facilidad en el mundo de la computación al desarrollarse el lenguaje **FORTRAN** (**FOR**mula **TRAN**slation), que tal como su nombre indica, permite la introducción de fórmulas que luego son traducidas a código máquina mediante un compilador desarrollado por el equipo de IBM en 1957.

Posteriormente nacen lenguajes de alto nivel como **LISP** (1958) y **COBOL** (1959). Éstos permitían una abstracción que brindaba una vida más fácil al programador, dejando los detalles específicos de la máquina para los ensambladores y compiladores.

Thomas Kurtz y **John Kemeny** desarrollan, en 1964, el lenguaje **BASIC** (**B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode) cuyo objetivo era servir a la enseñanza de la programación. En 1968, **Niklaus Wirth** crea **Pascal**.

Comienzan a aparecer los **paradigmas de la programación**, es decir, filosofías y enfoques diferentes según sea el lenguaje, para resolver problemas. Un artículo de 1966 publicado por **Corrado Böhm** y **Giuseppe Jacopini** sienta las bases del paradigma estructurado. Donde según su **teorema del programa estructurado**, establecen que toda rutina puede implementarse en un lenguaje de programación que solo combine tres estructuras lógicas: **secuencia**, **selección** e **iteración**.

Esto vino a solucionar grandes problemas que había para la época con la sentencia **GOTO** (instrucción que permitía transferir el control a un punto determinado del código), ya que la misma daba grandes dificultades a la hora del seguimiento y la corrección de los programas. En 1968, **Edsger Dijkstra** escribe un artículo donde muestra las contras de emplear esta sentencia y desalienta su utilización.

La década del 70 marcó hitos importantes con el desarrollo del lenguaje **C**, estructurado, por **Dennis Ritchie** en los Laboratorios Bell entre 1969 y 1972. Aparecen otros paradigmas, como la **programación lógica** con el lenguaje **Prolog** en 1972, un dialecto de Lisp llamado **Scheme (programación funcional)** en 1975 y a finales de la década, el primer lenguaje de **programación orientada a objetos**, llamado **Smalltalk**.

En los 80s, se profundizó en el estudio de los paradigmas existentes. El paradigma estructurado mostraba dificultades con el empleo de variables globales, por lo que la programación orientada a objetos comienza a ser estudiada con mayor interés. En 1980, se crea un lenguaje **C con clases**, que en 1983 pasa a llamarse **C++**. Evoluciona el hardware con avances en los microprocesadores, haciéndolos más eficientes para los compiladores de lenguajes de alto nivel y ya no tanto para los programadores humanos de lenguaje ensamblador.

En la década del 90, Internet comienza a tener mayor auge, logrando abrir nuevos horizontes. Se crea **Haskell** en 1990. **Python** y **Visual Basic** ven la luz al año siguiente. Para esta época los sistemas operativos ya ofrecían las grandes ventajas de las **interfaces gráficas de usuario (GUI)**, por lo que los lenguajes de programación se adaptan a ello. El año 1995 vio nacer a **PHP**, **Java** y **JavaScript**, y con ello herramientas que mejoraban la productividad del programador, como los **entornos de desarrollo integrado (IDE)** con recolectores de basura y herramientas de depuración.

A partir del tercer milenio, Microsoft crea, en 2001, el lenguaje **C#**, basado en **Java** y lo incorpora a su plataforma **.NET**.

Las tendencias actuales muestran un auge del lenguaje **JavaScript**, ya que, con el avance de la infraestructura de las redes de comunicaciones, todo se orienta hacia la web. De todas maneras, **Java** continúa dominando gran parte del mundo empresarial. El lenguaje **C** sigue teniendo gran demanda en aplicaciones de sistemas embebidos.

Las grandes empresas de hoy invierten grandes capitales en **machine learning** e **inteligencia artificial**, para desarrollar aplicaciones que permitan analizar grandes cantidades de datos en busca de mejores soluciones y estrategias de marketing.

Programar no es inventar y razonar las cosas desde cero, sino reutilizar componentes de software ya creados para ensamblarlos y generar aplicaciones más robustas. Hoy en día hay presentes cientos de **frameworks** y **librerías** para diversos propósitos que abstraen y simplifican ciertas cuestiones al programador.

De todas maneras, lo que hoy es furor, mañana será obsoleto. Son las reglas de juego de esta industria. Tomalo como un desafío y nunca dejes de capacitarte.

¿Por dónde empezar?

Es importante entender que la informática tiene como objetivo automatizar el proceso de trata de información.

Los datos normalmente son brindados por el usuario vía dispositivos periféricos de entrada, como pueden ser el teclado, mouse, scanner, cámara, micrófono, etc. La computadora los procesa y genera una salida, vía dispositivos periféricos de salida, como la pantalla, parlantes, auriculares, impresora, etc.

Para poder comenzar a programar la computadora, es necesario conocer el concepto de **algoritmo**.

caemci

Algoritmos

Definición

Se trata de un conjunto de indicaciones finitas (me refiero, a que dichas indicaciones tienen un comienzo y un final) y ordenadas de forma lógica que permite la resolución de un problema dado.

Se atribuye el término al matemático y astrónomo persa **Musa al-Juarismi**, quien vivió entre los años 780 y 850.

Con seguridad, habrás leído e interpretado algoritmos sin que supieras que se trataba de ellos. Se me ocurren algunos ejemplos cotidianos con los que quizás hayas interactuado:

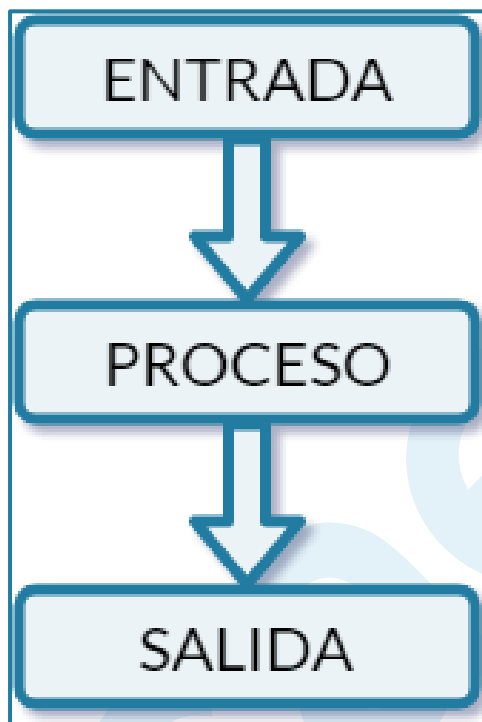


Ilustración 1: Componentes de un algoritmo.

- Un manual de instrucciones para colgar una televisión en la pared.
- Una receta de cocina para preparar un postre.
- Las directivas de un jefe a su empleado.
- Las indicaciones de un GPS para llegar a destino.
- Los pasos para calcular el cociente entre dos números enteros.

La importancia de los algoritmos en la informática es trascendental, dado que la programación tiene como objetivo la implementación de ellos en una computadora, para que sea ésta quien los ejecute y resuelva determinado problema, sin embargo, éstos trascienden la disciplina informática, pudiendo encontrarlos en la matemática o la lógica.

Características

Para que un algoritmo pueda ser considerado como tal, debe cumplir con las siguientes claves:

- Debe ser **preciso**, indicando el orden de realización de cada paso.

- Debe estar **definido**, obteniéndose el mismo resultado si se repite el proceso con los mismos datos de entrada.
- Debe ser **finito**, teniendo un número de pasos que permita llegar a un final.

Representación

Volviendo a los ejemplos de la página 12, podemos expresar los algoritmos en un lenguaje natural cuya intuición y comprensión resultan convenientes, pero con la desventaja de que son imprecisos.

La manera de representar algoritmos de manera precisa y sin ambigüedades es mediante dos maneras formales: como **diagrama de flujo** o como **pseudocódigo**.

Un **diagrama de flujo** es una representación gráfica de un algoritmo. Seguramente te resulta familiar, pues es usado también para describir procesos industriales o de negocio. Son bastante convenientes al principio, dado que son bastante fáciles de entender, gracias a que presentan las instrucciones de una manera gráfica.

La clave para entenderlos es comprender que existen diferentes componentes, cada uno con una forma distinta. La unión de ellos permite formalizar una solución:

Forma	Descripción
	Delimitador. Representa el comienzo o la finalización de la secuencia de instrucciones.
	Entrada. Representa la adquisición de datos por medio de un periférico, como el teclado.
	Proceso. Representa una operación o una tarea.
	Salida. Representa la visualización de los resultados por medio de un periférico, como la pantalla.
	Condición. Representa un interrogante cuya evaluación debe dar únicamente VERDADERO o FALSO .
	Línea de flujo. Representa la dirección de la secuencia de instrucciones.

A continuación, te presento un diagrama de flujo que muestra cómo preparar café:

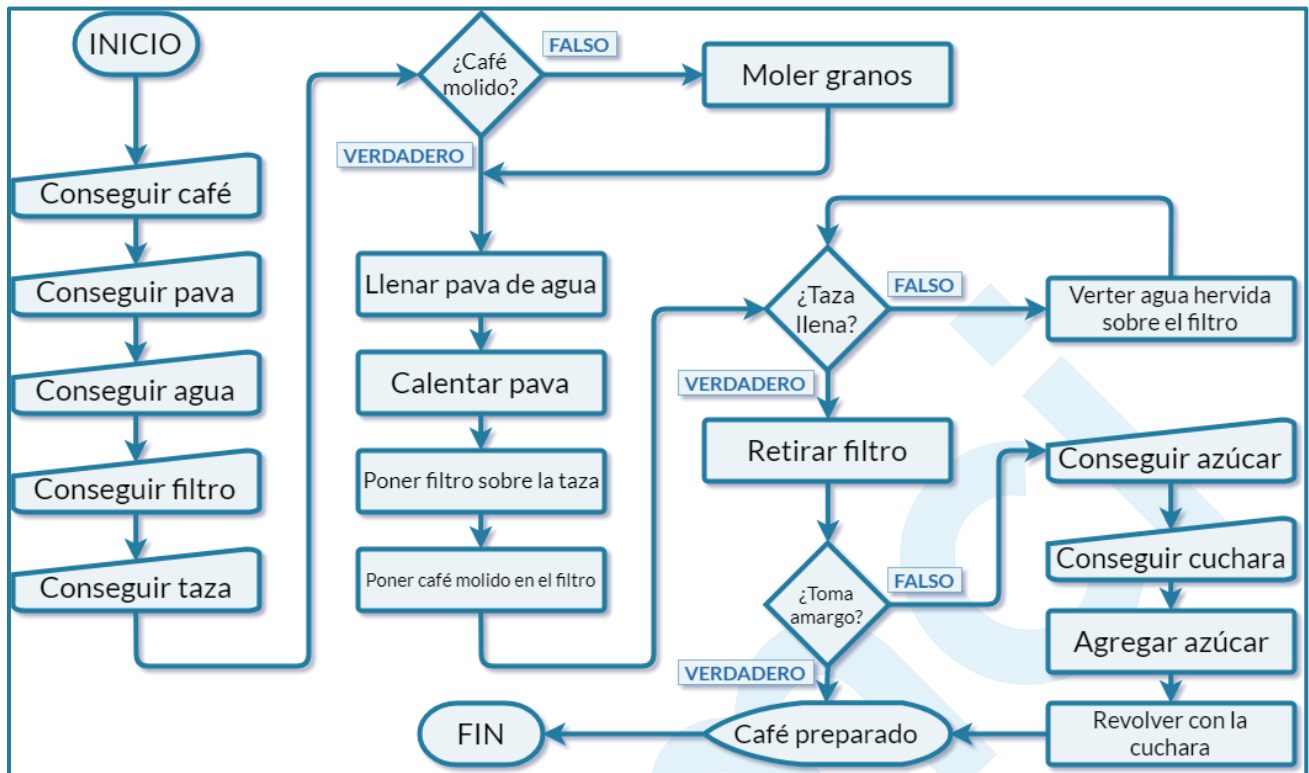


Ilustración 2: Diagrama de flujo de cómo preparar café

La desventaja de los diagramas de flujo es que una computadora no puede interpretarlos directamente. Son simples representaciones gráficas para los seres humanos.

La manera más cercana a la máquina de representar un algoritmo, pero todavía interpretable fácilmente por una persona, es mediante **pseudocódigo**.

Un **pseudocódigo** es una descripción de las instrucciones de manera tal de ser muy similar al formato que se obtiene al utilizar un lenguaje de programación. El punto es que el pseudocódigo no tiene un estándar de reglas sintácticas a seguir, sino que es constituido por convención por uno o más programadores para tener una solución abstracta del problema, algo así como una base para luego transcribir ese algoritmo en un lenguaje de programación real.

A continuación, te presento un posible pseudocódigo correspondiente al problema de preparar café, cuyo diagrama de flujo ya se detalló en la ilustración anterior:

1	INICIO
2	CONSEGUIR café
3	CONSEGUIR pava
4	CONSEGUIR agua
5	CONSEGUIR filtro

6	CONSEGUIR taza
7	SI el café no está molido
8	MOLER café
9	LLENAR pava CON agua
10	CALENTAR pava
11	PONER filtro EN taza
12	PONER café EN filtro
13	MIENTRAS la taza no esté llena
14	VERTER agua EN filtro
15	RETIRAR filtro
16	SI no toma amargo
17	CONSEGUIR azúcar
18	CONSEGUIR cuchara
19	AGREGAR azúcar
20	REVOLVER CON cuchara
21	SERVIR café
22	FIN

Código 1: Cómo preparar café.

Preparando el ambiente

Tal como mostré anteriormente, los algoritmos son independientes de cualquier medio y lenguaje, por eso, mi propuesta es verter los primeros conceptos de programación mediante diagramas de flujo y pseudocódigo. Es clave asimilar los fundamentos de la programación antes de pasar a un lenguaje formal de programación, donde cada uno puede tener sus bemoles.

Existe un software cuyo objetivo es promover la enseñanza de los fundamentos de la programación llamado **PSeInt** (abreviatura para **Pseudo Intérprete**), el cual utilizaré a lo largo de este libro.

Según el autor, *“PSeInt es una herramienta para asistir a un estudiante en sus primeros pasos en programación. Mediante un simple e intuitivo pseudolenguaje en español (complementado con un editor de diagramas de flujo), le permite centrar su atención en los conceptos fundamentales de la algoritmia computacional, minimizando las dificultades propias de un lenguaje y proporcionando un entorno de trabajo con numerosas ayudas y recursos didácticos”*.

Instalación de PSeInt

Para proceder a la instalación de PSeInt, descargá el instalador desde la siguiente web: <http://pseint.sourceforge.net/index.php?page=descargas.php>

El software está disponible para varias plataformas, aunque mis indicaciones serán para **Windows**, dado que es el sistema operativo más utilizado. Una vez descargado el instalador, ejecutalo para que aparezca la siguiente ventana:

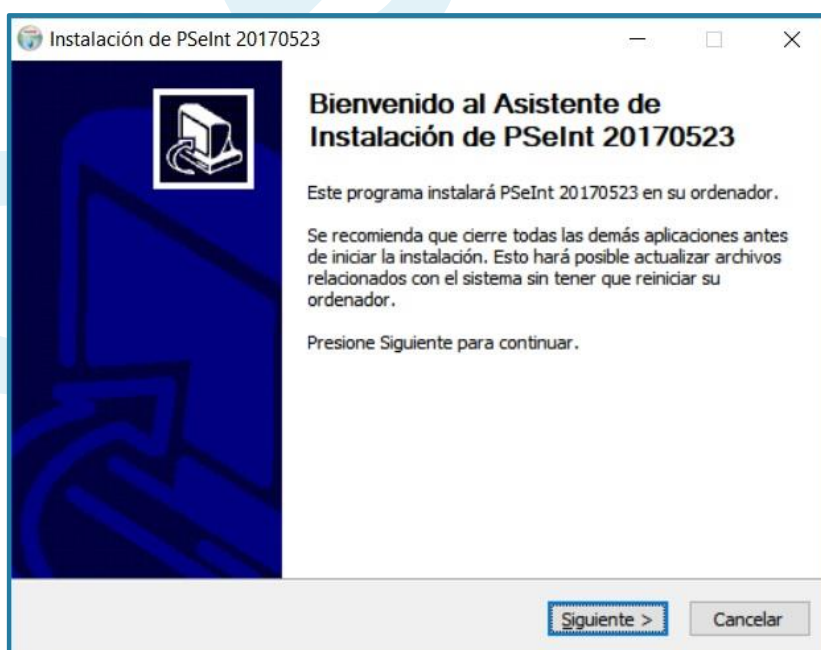


Ilustración 3: Ventana inicial del instalador.

Presioná **Siguiente**. A continuación, aparecerá el acuerdo de licencia:

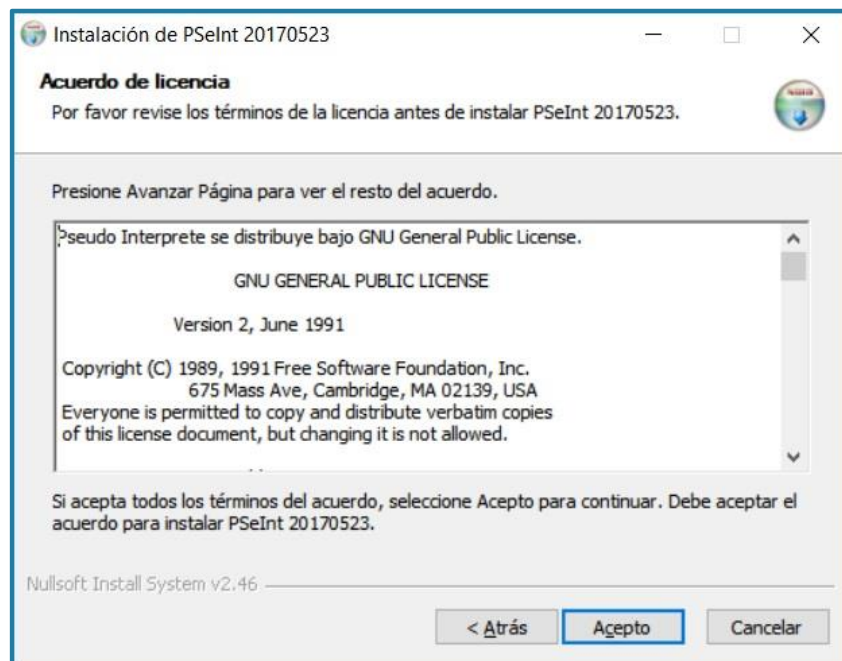


Ilustración 4: Acuerdo de licencia de PSeInt.

Si estás de acuerdo, presioná **Acepto**. A continuación, aparecerá la selección del directorio de instalación, el cual por defecto es **C:\Program Files (x86)\PSeInt**. Si deseás modificarlo, debés hacer click en **Examinar**.

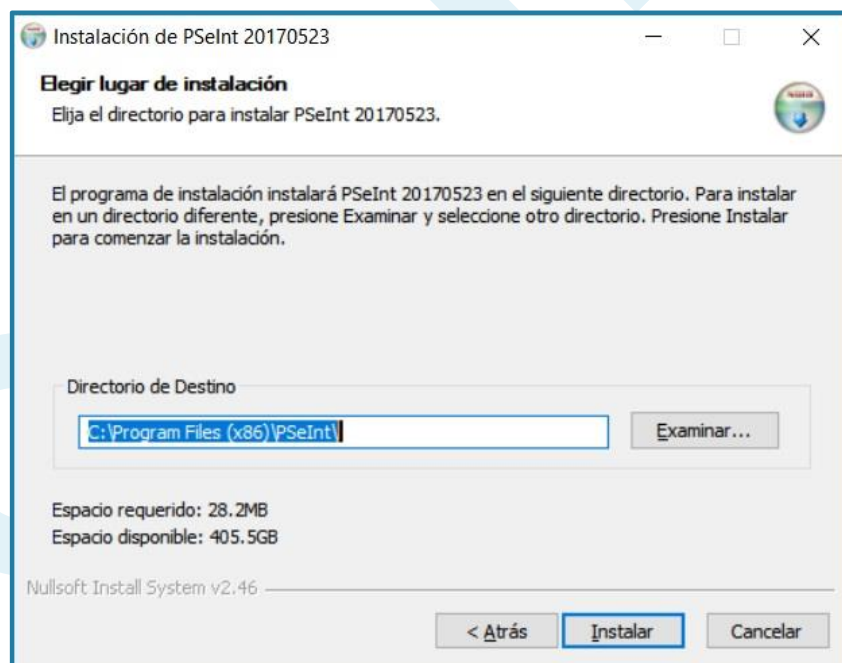


Ilustración 5: Directorio de instalación de PSeInt.

Hacé click en **Instalar** para que comience la copia de archivos. Una vez finalizada, aparecerá la última ventana:

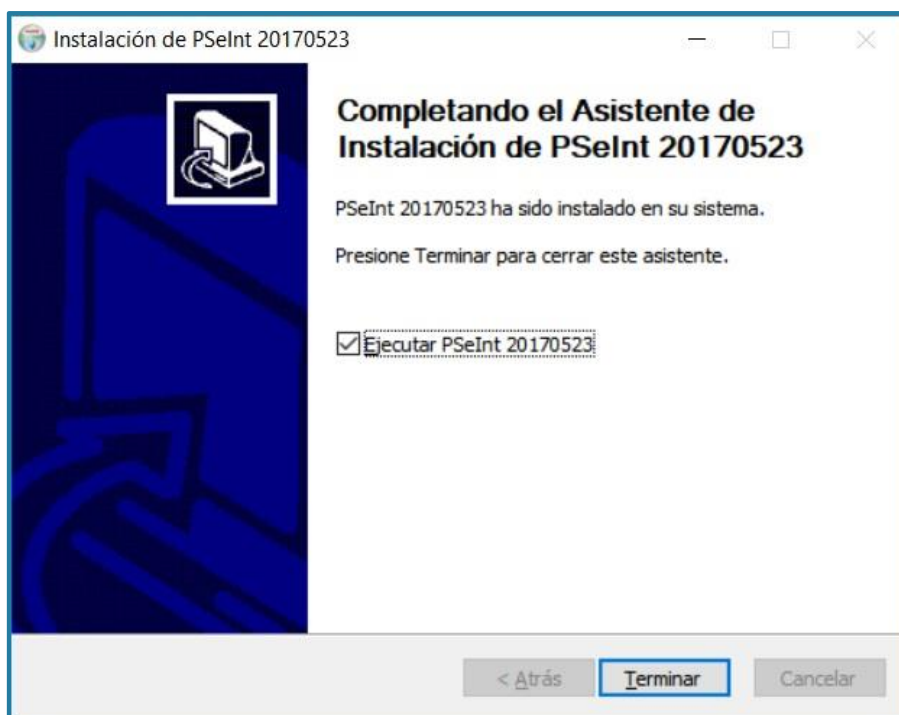


Ilustración 6: Ventana final del instalador de PSeInt.

Dejá tildada la opción de **Ejecutar** y presioná **Terminar** para que se abra el programa.

Configuración de la sintaxis

PSeInt trabaja con un pseudolenguaje flexible y personalizable a gusto del docente o estudiante. Es importante establecer la misma sintaxis que uso en este libro para poder llevar a cabo las prácticas de manera satisfactoria.

Para configurar la sintaxis del pseudolenguaje, debés ir al menú **Configurar** y elegir **Opciones del Lenguaje (perfiles)...**

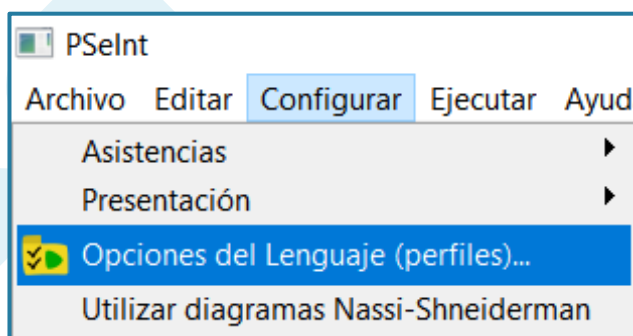



Ilustración 7: Menú Configurar.

Allí aparecen listados distintos centros educativos, cada uno con su sintaxis predefinida. Como no está listado el perfil que usaré en este libro, hacé click en el

botón  **Personalizar...** que se encuentra debajo a la izquierda y asegurate que los

parámetros del lenguaje estén marcados exactamente igual que en la siguiente ilustración:

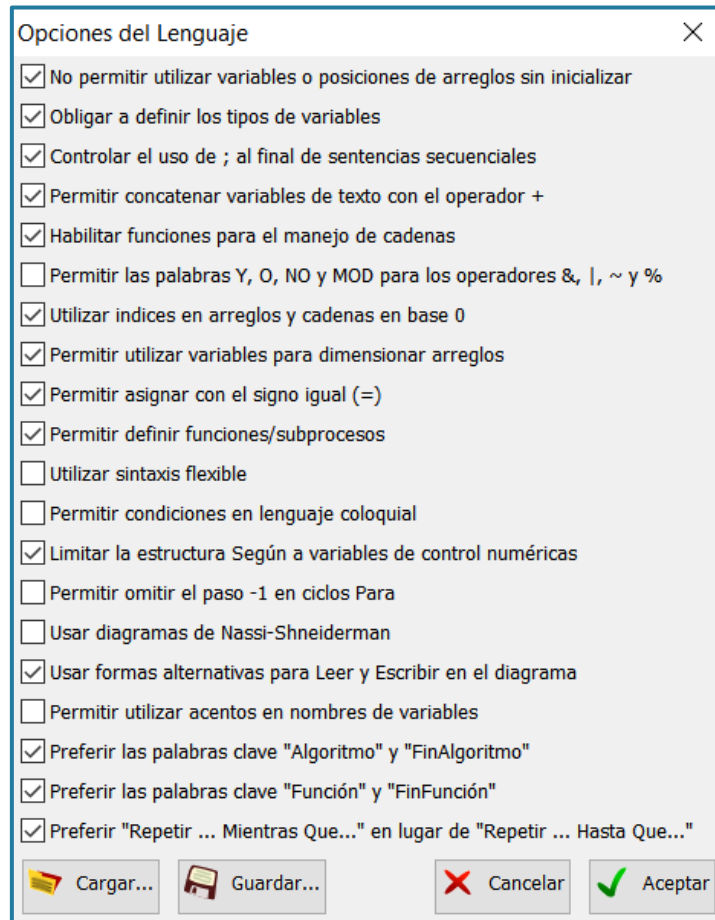


Ilustración 8: Parámetros del lenguaje usado en este libro.

Una vez terminado, **aceptá todas las ventanas**. El perfil quedará guardado hasta que se modifique, por lo que este proceso se realiza solo la primera vez.

Primer programa

La interfaz de PSeInt es bastante intuitiva. En el centro de la ventana vemos el editor de código para escribir las instrucciones a ejecutarse.

Para comprobar que todo está en orden, vas a realizar tu primer programa. Dicho sea de paso, todo estudiante que inicia en el mundo de la programación realiza el famoso **"Hola Mundo!"**.

Dentro de las sentencias **Algoritmo** y **FinAlgoritmo**, escribí literalmente la siguiente instrucción:

Escribir "Hola Mundo!";

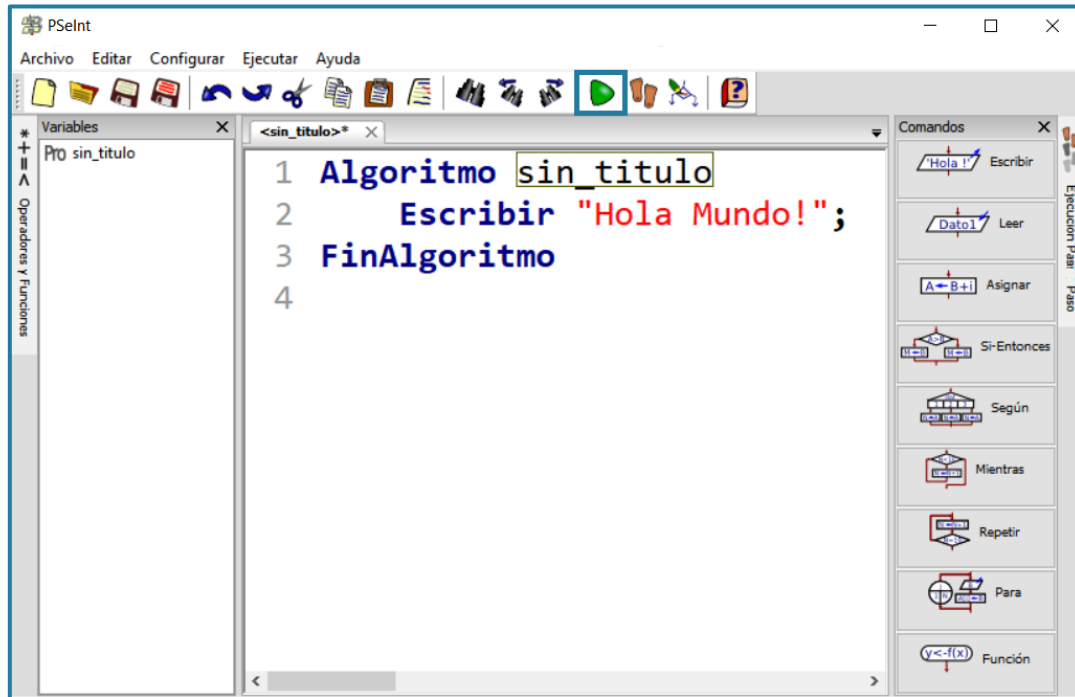



Ilustración 9: Editando el primer programa.

Si todo va bien (PSeInt no marca errores), estás en condiciones para ejecutar tu programa. Hasta acá, lo que ves no es más que un texto que sigue una sintaxis particular. La magia ocurre cuando PSeInt hace honor a su nombre e interpreta el pseudocódigo. Presioná el botón  o la tecla **F9**.

A continuación, se abre una nueva ventana llamada **consola**, con el resultado del programa que, tal como lo escribiste, es un mensaje **"Hola Mundo!"**, sin las comillas, lo cual es correcto.

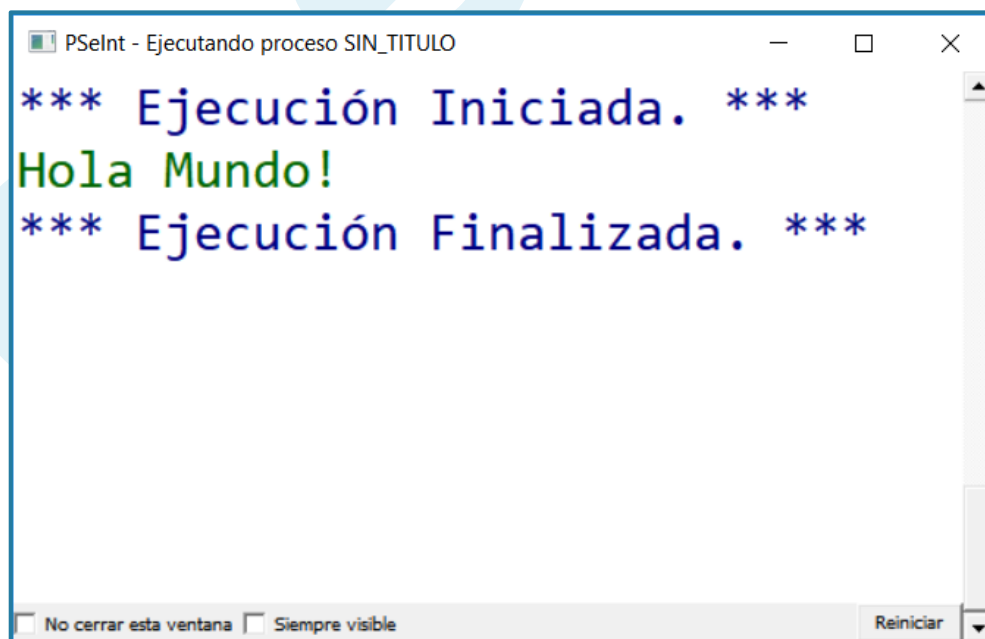


Ilustración 10: Ejecución del primer programa.


Entiendo tu posible frustración. Quizá esperabas que tu primer programa fuera un juego de guerra o un software de facturación. Te pido paciencia.


Hoy en día el software tiene como componente fundamental la **GUI** (siglas en inglés para **Interfaz Gráfica de Usuario**). Es difícil imaginar un programa sin botones, menús, ventanas emergentes, barras deslizables, colores, etc.

Lo que realizarás en este libro son **programas de consola**, es decir, las entradas y salidas no serán mediante componentes gráficos, sino a través de la consola o terminal del sistema.

No es poco lo que puede hacerse en la consola al principio. Lo importante es sentar las bases y fundamentos de la programación para que luego puedas encarar otros paradigmas y conceptos que te permitan construir software con ventanas gráficas.

Todos alguna vez comenzamos por acá, así que, **¡ánimo, ya irás avanzando!**

Antes de continuar, sería bueno que vayas guardando tus pseudocódigos para repasarlos o ejecutarlos en otro momento. A través del botón , podés guardar el pseudocódigo en un archivo, cuya extensión es **.psc**, simplemente para identificar que se trata de un código de PSeInt, aunque en realidad es un archivo de texto plano que puede abrirse con cualquier editor, como por ejemplo, el bloc de notas.

Para abrir un pseudocódigo ya guardado, debés presionar el botón  y buscarlo entre tus archivos.

Errores

Antes de continuar, quiero hacerte una aclaración sobre algo que a menudo ocurrirá: **el error**. No te preocupes, es parte del proceso y hasta los programadores más experimentados no son ajenos a ellos. ¡Somos humanos!

Los errores que se pueden cometer se dividen en dos tipos: **errores en tiempo de compilación** y **errores en tiempo de ejecución**.

Los **errores en tiempo de compilación** son aquellos en los que, cuando el compilador de código detecta que algo no está bien, acusa un mensaje donde describe tal error, haciendo que el programa no pueda si quiera ejecutarse. PSeInt no es compilado, sino interpretado, por lo que la definición anterior no es del todo cierta para este tipo de lenguaje, por ello es que en este caso usaré para el término **errores de sintaxis**.

Para resumir, los **errores de sintaxis**, como, por ejemplo, que a la instrucción:

Escribir "Hola Mundo!";

le quites el punto y coma, harán que no puedas siquiera ejecutar el programa. Probalo y verás.

Los **errores en tiempo de ejecución** son aquellos que se producen cuando el programa ya ha sido ejecutado sin errores sintácticos. En determinado momento, el programa detectará un error y no podrá continuar, ocasionando que finalice de forma abrupta.

Hay errores que no impiden que el programa se ejecute, pero provocan que los resultados quizá sean inesperados. Son **errores lógicos**, y son los más difíciles de detectar, dado que se requiere volver a analizar y probar el código en busca de la falla.

A lo largo de los temas que se irán desarrollando, te mostraré los errores más típicos que se pueden cometer. Recordá que del error es de donde más y mejor se aprende.

caemci

Flujo secuencial

Como primera medida, debés entender que las instrucciones se ejecutan de manera natural una tras otra, en el orden en que fueron escritas. Este flujo se denomina secuencial y es el más sencillo de todos. La máquina interpretará y ejecutará paso a paso las líneas, desde la primera hasta la última.

Observá el siguiente diagrama de flujo:

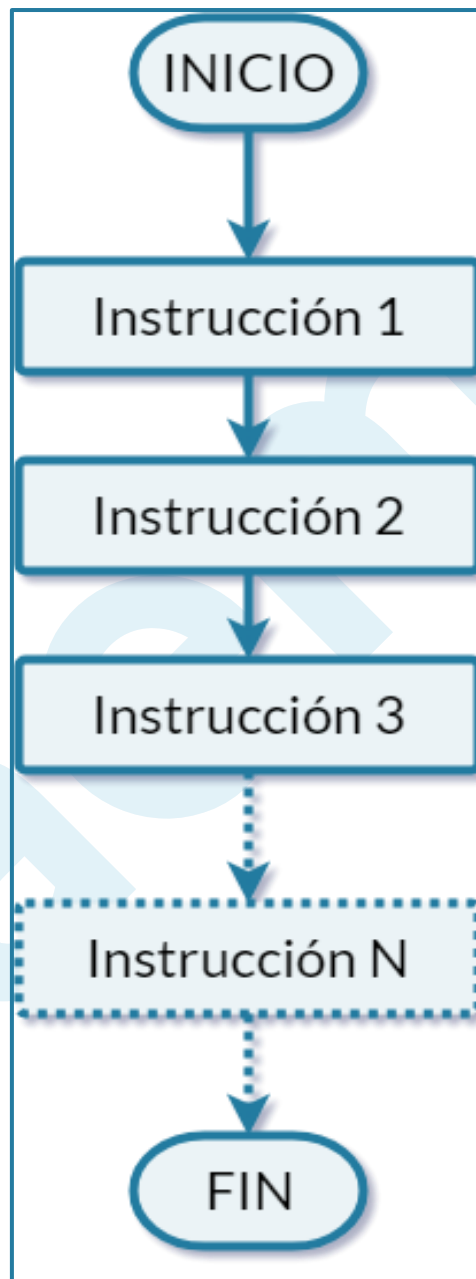


Ilustración 11: Flujo secuencial.

No hay una regla que diga de cuántas instrucciones deba constar un programa, sino que depende del problema a resolver.

Instrucción de salida

Lo primero que vas a aprender es a mostrarles mensajes al usuario, en este caso, a vos mismo/a. Si bien ya la hemos utilizado en nuestro **Primer programa**, conviene enfatizar sobre la instrucción que lo permite.

La sintaxis es la siguiente:

Escribir <expresión>;

- **Escribir** se tipea literalmente, indica que queremos mostrarle un mensaje al usuario a través de la pantalla.
- La palabra <expresión> indica que en ese lugar debe ir una expresión válida. Puede ser una palabra, un número o una operación matemática.
- El **;** es obligatorio e indica que finalizó la instrucción.


Vamos a hacer un programa que le muestre al usuario la frase **"Quiero aprender programación"**.

```
1 Algoritmo salida
2     Escribir "Quiero aprender programación";
3 FinAlgoritmo
```

Código 2: Algoritmo con instrucción de salida.

Para el intérprete, es indistinto el uso de mayúsculas y minúsculas. Observá en tu editor que la palabra **Escribir** se colorea al igual que el **;**. El texto entre comillas adopta otro color, indicando que no se trata de una instrucción, sino de una cadena de caracteres.

No olvides encerrar tu frase entre comillas dobles y cerrar la sentencia con punto y coma. Tené cuidado de no copiar los números de línea de la izquierda, solo copió el código.

Asumo que ya sabés ejecutar un programa, pero lo recuerdo una vez más: presioná el botón  o la tecla **F9**.

El resultado es un programa con una sola salida, que fue lo que escribiste. Observarás que sale sin las comillas. Esto es porque las comillas sirven para indicarle al intérprete el comienzo y el fin de una palabra o frase.

¿Qué sucede si volvés a escribir debajo otra instrucción **Escribir**? Recordá que el intérprete ejecuta las instrucciones de forma secuencial, por lo que verás la expresión del primer **Escribir** seguido de otra línea con la expresión del segundo **Escribir**.

El no uso de comillas para una cadena o algún caso de ambigüedad (que abran comillas y que no cierren, o que no abran y sí cierren) derivará en un error de sintaxis.

Probá los resultados del siguiente programa:

```
1 Algoritmo doble_salida
2     Escribir "Quiero aprender programación";
3     Escribir "Tengo un gran entusiasmo :D";
4 FinAlgoritmo
```

Código 3: Algoritmo con dos instrucciones de salida.

Una manera de alterar la posición del cursor de salida es usar una palabra clave llamada **Sin Saltar**. La sintaxis es la siguiente:

Escribir Sin Saltar <expresión>;

Toda sentencia **Escribir** que contenga **Sin Saltar**, hace que el cursor quede posicionado al final de la línea y no en una nueva, lo que provoca que la próxima expresión que hubiere en una sentencia **Escribir** se coloque en la línea actual, a continuación de la expresión anterior.

Probá el siguiente programa:

```
1 Algoritmo salida_sin_salto
2     Escribir Sin Saltar "Quiero aprender programación";
3     Escribir "Tengo un gran entusiasmo :D";
4 FinAlgoritmo
```

Código 4: Algoritmo con salida sin salto de línea.

El resultado es un mensaje por pantalla con la siguiente leyenda:

"Quiero aprender programaciónTengo un gran entusiasmo :D"

por supuesto, sin las comillas.

Hasta aquí, has mostrado texto por consola al usuario de tu programa. ¿Se podrán mostrar números? Claro que sí. Y hasta expresiones aritméticas, ya que, ante todo, la computadora es una calculadora. Los números y las expresiones van sin comillas.

Probá el siguiente programa:

```
1 Algoritmo salida_expr_numericas
2     Escribir 256;
3     Escribir 3.14;
```

```
4      Escribir 1 + 1;  
5  FinAlgoritmo
```

Código 5: Algoritmo con salida de expresiones numéricas.

Verás en líneas separadas y en el mismo orden en que lo programaste los números 256, 3.14 y 2. No hay mucho que decir de la primera sentencia. Simplemente escribiste el número 256. En la segunda hay que recordar que el **separador decimal es el punto y no la coma**. En el tercer caso, el resultado es 2 porque el **intérprete evalúa la expresión y obtiene un resultado que finalmente envía a la pantalla**.

Antes del próximo tema, realizá esta prueba:

```
1  Algoritmo salida5  
2      Escribir "8";  
3      Escribir 8;  
4  FinAlgoritmo
```

Código 6: Diferencia entre datos numéricos y alfanuméricos.

El resultado es la salida del número 8, dos veces. Quizá tu conclusión sea que poner los números entre comillas y sueltos sea equivalente, cosa que es un grave error. Enseguida verás por qué.

Tipos de datos

Pongamos algo en claro: **para la computadora cualquier dato es un número binario**. La diferencia es, a alto nivel, cómo el programador interactúa con estos datos.

Hay lenguajes de programación para los cuales no hay discriminación de datos. La palabra "Hola" puede ser mezclada y operada con el número 4 o el carácter 'z'. A estos lenguajes, se los llama **débilmente tipados**.

Otros lenguajes ponen énfasis en la diferencia entre los datos, no es lo mismo un número entero que uno real, tampoco un carácter simple que una cadena de caracteres. Los lenguajes que siguen tal descripción se denominan **fuertemente tipados**.

Como la mayoría de las cosas, no existe una manera mejor que otra. Personalmente prefiero los lenguajes fuertemente tipados, dado que, si bien parecen más dificultosos, me permiten tener un mayor control como programador y además facilitan la depuración (seguimiento y corrección de errores). De todas maneras, hay que tener en cuenta que no siempre es posible esta elección.

Básicamente, existen dos claros tipos de datos diferentes: los **numéricos**, los **alfanuméricos** y los **booleanos**.

Datos numéricos

Cualquier dato que represente un número, positivo, negativo, fraccionario o el cero, es considerado un **dato numérico**.

Los lenguajes fuertemente tipados hacen una clasificación aún dentro de esta categoría. No es lo mismo un número entero como el **12** que el número **459213**. Recordá que la computadora guarda los datos de forma binaria. Cuando ves un **12** en la pantalla, en realidad la máquina lo representa como un **1100** en binario. Y el número **4592193** en binario se representa así: **1110000000111001101**.

Como verás, guardar el número **459213** requiere más espacio en memoria que el número **12**. Si me pongo un poco más técnico, supongamos que cada celda de una memoria aloja **1 byte** (o sea **8 bits**, o sea, **8 dígitos binarios**). Notarás que el número **12** cabe perfectamente en una celda, de hecho, sobran cuatro dígitos que se rellenan con ceros a la izquierda. En cambio, el número **459213** necesita 19 dígitos, lo cual demanda 5 celdas de memoria. Por este concepto es que los lenguajes fuertemente tipados discriminan entre **enteros cortos** (en inglés, **short**), **enteros normales** (en inglés, **int**) y **enteros largos** (en inglés, **long**), etc. Si el programador tiene en cuenta con qué datos numéricos va a trabajar y hace uso de los tipos correctos, el programa será más eficiente.

¿Qué sucede con los números no enteros? A estos se lo denominan, **números de coma flotante o de punto flotante**. También se guardan en binario, de una forma un poco más compleja que no merece la pena que explique. Que un número real ocupe más o menos espacio depende no solo de la parte entera sino también de la cantidad de dígitos decimales. Algo que cabe destacar es que no existen dígitos decimales infinitos. El número **1/3** que en número decimal es un **0.3333...** (el separador decimal en la informática es el punto) tiene infinitos dígitos **3** en la teoría, pero en la práctica, la computadora guardará un número finito de ellos. Por eso muchos lenguajes fuertemente tipados diferencian entre **números de coma flotantes simples o dobles**, según la cantidad de cifras decimales que se pueden guardar. Si al número **1/3** lo guardamos como número flotante doble en vez de simple, tendremos más precisión en cuanto al número real a costa de ocupar el doble de celdas de memoria. La elección depende del problema a resolver.

Tampoco es lo mismo un **12** que un **12.0** ¿Te estoy mareando? No es la idea. Claro que el **12** y el **12.0** son el mismo número, sin embargo, la computadora los trata como tipos diferentes. Un **12** es un número entero, pero sabemos que todo número entero es un número real, por lo que, si se guarda el **12** como número de coma flotante, ocupará más memoria, pero permitirá operar con otros números de coma flotante.

Datos alfanuméricos

El otro conjunto de datos a analizar es el de los alfanuméricos. En esta categoría se encuadran los caracteres como las letras del alfabeto o los símbolos.

Repito, la computadora guarda los datos de manera binaria. Cómo la máquina codifica los caracteres merece explicarlo en un capítulo aparte, pero digamos que tal vez hayas escuchado hablar de **ASCII**, acrónimo inglés de **American Standard Code for Information Interchange** (Código Estándar Estadounidense para el Intercambio de Información). Este código representa una tabla donde cada carácter tiene asignado un número. ¿Alguna vez has visto que si mantenés presionada la tecla **ALT** y teclás el número **64** en el teclado numérico, se escribe un carácter "@" (arroba)? Tiene que ver con este código, que contiene 128 caracteres (los primeros 32 de control, no imprimibles) y cada uno de ellos ocupa 1 byte. En la versión extendida se ofrecen 128 caracteres extra.

Caracteres imprimibles					
Número	Símbolo	Número	Símbolo	Número	Símbolo
32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	\$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(72	H	104	h
41)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[123	{
60	<	92	\	124	
61	=	93]	125	}
62	>	94	^	126	~
63	?	95	_	127	DEL

Ilustración 12: Tabla de caracteres ASCII.

Un tipo de dato un poco más complejo que merece ser mencionado es el de la cadena de caracteres (en inglés, **string**). Una cadena es un conjunto de caracteres que permiten formar una palabra, frase o párrafo. Desde la palabra **"pez"** hasta la biblia completa se considera una cadena de caracteres. Este tipo de dato se diferencia a los vistos hasta ahora en que no es considerado un tipo de dato primitivo, sino una estructura más compleja formada por tales. En realidad, una cadena es un vector o arreglo (array) unidimensional de caracteres. Hay lenguajes que las manejan como tal mientras que otros intentan emularlo como un dato primitivo para facilitar su tratamiento.

Verás que los números también están dentro de la tabla ASCII y es aquí donde se responde la pregunta pendiente en la página 26: **El número 8 sin comillas es un número entero**, capaz de ser operado matemáticamente. **El número '8' como carácter** (en muchos lenguajes, se lo encierra **entre comillas simples**) es tratado como un símbolo. No sirve para operar matemáticamente, sino para ser concatenado con otros caracteres y así formar palabras. Inclusive podemos tener **un "8" como cadena de caracteres** (en muchos lenguajes, se lo encierra **entre comillas dobles**), es decir, una estructura de datos de un símbolo de longitud. Es importante tener en cuenta estas tres diferencias.

Datos booleanos

Los booleanos son un tipo especial de dato que representan un valor de verdad según la lógica clásica (veremos un poco de ella más adelante). Deben su nombre a **George Boole** (1815 - 1864), matemático y lógico británico enunciador del álgebra que lleva su nombre.

En un tipo de dato booleano solo caben dos valores posibles: **VERDADERO** (en inglés, **true**) o **FALSO** (en inglés, **false**). Algunos lenguajes pueden representarlo como **0** para **FALSO** y **1** para **VERDADERO**, aunque es importante destacar que **un 0 booleano no es igual a un 0 entero. Lo mismo para el 1**. De hecho, si trabajamos con álgebra de números el resultado de **1 + 1** es **2**, en cambio, en el álgebra de Boole, **VERDADERO + VERDADERO** da como resultado **VERDADERO**.

Los datos booleanos son los más económicos en cuanto a espacio en memoria, dado que se pueden representar con tan solo un bit. Pero como cada celda de memoria guarda 8 bits y no se puede fraccionar, termina ocupando 1 byte.

Convención

Los tipos de datos anteriormente descriptos y su espacio ocupado en memoria son una aproximación abstracta según los casos más típicos. Cada lenguaje de programación establece cuántos, cuáles y qué costos (me refiero a lugar en memoria) tiene cada tipo de dato.

La sintaxis de PSeInt que te he hecho importar establece al intérprete para que se maneje de forma fuertemente tipada. Vamos a adoptar una convención, que es contar con 4 tipos diferentes de datos detallados a continuación:

- **ENTERO.** Permite trabajar con números enteros comprendidos en el rango entre **-2147483648** y **2147483647**.
- **REAL.** Permite trabajar con números de coma flotante comprendidos en el rango entre **2.2E-308** y **1.8E+308**.
- **LOGICO.** Permite trabajar con valores booleanos (**VERDADERO** o **FALSO**).
- **CADENA.** Permite trabajar con conjuntos de caracteres.

Es importante que tengas presentes estos tipos de datos para los siguientes ejemplos.

PSelnt trata de igual manera a los caracteres simples que a las cadenas. Una cadena puede ser encerrada entre comillas dobles `" "` o mediante comillas simples `' '`, de forma indistinta. Mi recomendación es que encierres a las cadenas con comillas dobles. Muchos lenguajes formales reservan el uso de comillas simples para referirse a caracteres simples. Un carácter simple no se trata de la misma manera que una cadena de caracteres.

Expresiones aritméticas

Has comprobado que la computadora evalúa automáticamente cualquier expresión aritmética, proveyendo el resultado. En este apartado, voy a formalizar dicho descubrimiento.

Una expresión aritmética es aquella que se compone de números y operadores aritméticos, siguiendo una lógica. Es importante aclarar que toda expresión aritmética da como resultado un número y creeme que no está de más aclararlo, ya que más adelante veremos expresiones que retornan otro tipo de dato.

Para poder formular expresiones aritméticas, es necesario conocer los operadores aritméticos.

Operadores aritméticos

Consta de los operadores fundamentales de la aritmética con el agregado de la potencia y el módulo o residuo, que devuelve el resto entero que se produce al realizar un cociente entre dos números enteros.

Operador	Nombre	Ejemplo	Resultado	Descripción
+	Suma	12 + 3	15	Devuelve la suma de dos expresiones.
-	Resta	12 - 3	9	Devuelve la resta de dos expresiones.

*	Multiplicación	12 * 3	36	Devuelve el producto de dos expresiones.
/	División	12 / 3	4	Devuelve el cociente de dos expresiones.
^	Potenciación	12 ^ 3	1728	Devuelve la potencia entre dos expresiones.
%	Módulo o Residuo	12 % 3	0	Devuelve el resto del cociente entre dos enteros.

Siguiendo los fundamentos de la aritmética, los operadores de multiplicación, división y módulo tienen mayor prioridad que la suma y resta. Así, por ejemplo, la expresión $2 + 3 * 4$ devuelve como resultado 14, pues primero se evalúa el término $3 * 4$ que resulta 12, y luego se realiza la suma entre 2 y 12.

Expresión vs. Instrucción

Es importante aclarar la **diferencia entre una expresión y una instrucción**.

Una expresión es un valor. Por más que veas una ecuación gigante llena de operadores aritméticos, la computadora va a resolverla y con ello obtendrá un único resultado. Más adelante verás que existe otro tipo de expresiones llamadas booleanas que devuelven un valor lógico.

Las expresiones por sí solas no sirven demasiado si no van acompañadas de una instrucción. ¿Qué es lo que querés hacer con ese resultado? ¿mostrarlo? ¿guardarlo?

Poner en una línea una expresión aritmética sin más, derivará en un error de sintaxis.

Por ello toda expresión debe ir acompañada de una instrucción. Hasta ahora solo conocés la instrucción de salida llamada **Escribir**, que te recuerdo tiene esta sintaxis:

Escribir <expresión>;

Por lo que hasta ahora podrías hacer un programa cuya línea conste de lo siguiente:

Escribir 2 + 3 * 4;

Cuya ejecución hará que por la pantalla se visualice el número 14.

¿Pero qué sucede si quiero hacer un cálculo aritmético para mostrarlo luego, o porque estoy haciendo cálculos de manera parcial? No es necesario, ni está bien mostrar resultados todo el tiempo. Es decir, si mi programa tiene 15 líneas, no significa que quiera escribir 15 líneas por pantalla al usuario. Éste quiere ver el resultado final, no el proceso, a no ser, claro está, que lo hagamos a propósito.

El siguiente nivel es ver una herramienta que te permitirá guardar tantos datos como quieras en la memoria de la máquina: **las variables**.

Variables

Definición

Una variable es un espacio reservado en la memoria de la máquina que permite alojar información. Tal como su nombre lo indica, esta información puede cambiar en el transcurso del programa, a contraposición de lo que sería una constante.

En los lenguajes débilmente tipados no es necesario declarar el tipo de información. Solo se reserva una porción de memoria y se asigna un dato.

En los lenguajes fuertemente tipados, a la hora de reservar, es necesario especificar el tipo de dato a guardar. Posteriormente, si se intenta alojar un tipo de dato determinado en una variable que fue declarada de otro tipo, se produciría un error.

¿En qué parte o qué dirección de memoria se aloja nuestro dato? No te interesa, ya que estás programando a alto nivel. En este caso lo que se utiliza es un identificador que permite referirse al dato en cuestión mediante un término mnemotécnico. Así es más fácil recordar que mis haberes de este mes se guardan en la variable que yo declaré **suelto** y no, por ejemplo, en la dirección de memoria **0x0A1B2C3D**.

Declaración

Al acto de reservar un espacio en la memoria se lo denomina **declaración o definición de una variable**. La gran mayoría de los lenguajes requieren que se reserven primero las variables que van a ser utilizadas a lo largo del programa.

Intentar utilizar una variable sin definir, derivará en un error en tiempo de ejecución.

Como buena práctica te sugiero declarar todas las variables al principio del programa, por más que haya alguna que vayas a utilizar a lo último. Esto permite una mejor legibilidad del programa.

Para definir una variable en PSeInt se utiliza la palabra clave **Definir** seguido de un nombre a tu elección, luego la palabra clave **Como** y por último el tipo de dato que alojará la variable. Recordá que esto es una instrucción, por lo que cierra con punto y coma. La sintaxis queda así:

Definir <identificador> Como <tipo_de_dato>;

Por ejemplo:

Definir edad Como Entero;

Al hacer esto, la computadora reservará un espacio disponible en la memoria que permitirá alojar un dato de tipo entero. A partir de este momento, cualquier referencia

a este dato, ya sea para cargarlo, modificarlo o leerlo es mediante el identificador que escogí, llamado **edad**.

En estos momentos, la variable **edad** no tiene valor. Ni si quiera un **0**. El **0** es un valor más como lo puede ser el **-49** o el **518**.

En el momento en que se declara una variable, esta tiene un valor indefinido. Cualquier intento de operar con este valor derivará en un error en tiempo de ejecución. A este caso se lo llama como **variable sin inicializar**.

Enseguida veremos cómo asignarle un valor a nuestra variable. Por ahora, quédate con el siguiente esquema:



Ilustración 13: Simulación de memoria con variables sin inicializar.

Como verás, cada variable puede ocupar una o más celdas según el tipo de dato del que fueron declaradas. Tampoco los identificadores se agrupan en el orden en que se declararon ni sus celdas tienen por qué ser contiguas (una al lado de la otra). Esto lo maneja el intérprete junto al sistema operativo. El programador de alto nivel se abstrae de estas cosas y trabaja con los nombres que eligió como identificadores.

¿Cuántas variables se pueden declarar? Si bien no es infinito, te aseguro que por ahora no debés preocuparte demasiado por ello. Podés reservar casi tantas como quieras.

```
1  Algoritmo declaracion_variables
2      Definir edad Como Entero;
3      Definir sueldo Como Real;
4      Definir hijos Como Entero;
5      Definir nombre Como Cadena;
6  FinAlgoritmo
```

Código 7: Definición de variables.

Reglas

Si bien la elección de un identificador para la variable es a elección del programador, deben tenerse en cuenta ciertas consideraciones:

- Un identificador debe comenzar con una letra.
- Puede contener letras, números y guiones bajos, pero no espacios, ni operadores.
- No puede coincidir con alguna palabra reservada del pseudolenguaje.
- No debe contener caracteres "extraños" como eñes, acentos o diéresis.

Las reglas las define cada lenguaje en particular, aunque la gran mayoría persigue los puntos antes citados.

Otra cuestión importante es que no puede haber identificadores repetidos, es decir, si ya declaraste la variable **suel**do y necesitas guardar otro, deberás declarar una segunda variable usando algún nombre alternativo como **suel**do2 o **suel**doB.

Definir una variable que ya lo había sido, derivará en un error en tiempo de ejecución.

Es muy importante para la legibilidad y depuración del código, tanto para vos como para un tercero, que utilices nombres para las variables que describan lo más claro posible el contenido de estas. Nombrar a todas las variables como `p1`, `x` o `abcd` no es muy conveniente.

Convención

Mientras sigas las reglas anteriores, no tendrás problemas de sintaxis con los identificadores de las variables.

Hay una convención bastante extendida y que te recomiendo fehacientemente que adoptes: **los nombres de las variables siempre van en minúsculas, inclusive su primera letra.**

No hay grandes problemas con nombres cortos, como `precio` o `apellido`, pero la lectura se dificulta con nombres compuestos por más de una palabra. Para ellos existe un estilo de escritura llamado **lowerCamelCase** (**lower** por comenzar con minúscula y **CamelCase** por la similitud de la notación con las jorobas de un camello) que consiste en comenzar el nombre de la variable con minúscula y a cada nueva palabra (sin espacio, no lo olvides) comenzarla con mayúscula. Con este estilo, se hace mucho más legible, por ejemplo, nombrar a una variable como `estadoDeCuenta` antes que `estadodecuenta`.

A partir de aquí, todas las variables que use en los ejemplos adoptarán la notación **lowerCamelCase**.

Operador de asignación

Hasta ahora has visto como declarar variables, las reglas de nombrado y la convención más difundida. Pero recordá que hasta ahora **las variables no están inicializadas**, es decir, su valor es **indefinido**.

Hay dos maneras de establecer un valor a una variable. La primera que veremos es a través del operador de asignación.

El operador de asignación de PSeInt es el `=`, aunque por defecto también puede usarse el `<-`, que es compuesto, dado que se conforma de dos caracteres. Es muy importante que no dejes un espacio entre ellos.

En este libro usaré el operador `=` dado que es el más común en la mayoría de los lenguajes formales y para tratar el tema de la confusión que genera con el operador de comparación que verás más adelante. Debido a que es fácil confundir al operador de asignación con el símbolo de igualdad usado en matemática, muchos lenguajes formales usan otros símbolos, como `<-` o `:=`.

La sintaxis para asignar un valor a una variable es la siguiente:

`<variable> = <expresión>;`

- La palabra **<variable>** indica que en ese lugar debe ir el identificador de una variable definida.
- **=** es el operador de asignación.
- La palabra **<expresión>** indica que en ese lugar debe ir una expresión válida, de un tipo de dato compatible con lo que espera guardar la variable.
- El **;** es obligatorio e indica que finalizó la instrucción.

Probá el siguiente programa:

```
1  Algoritmo carga_muestra_variables
2      Definir edad Como Entero;
3      Definir sueldo Como Real;
4      Definir hijos Como Entero;
5      Definir nombre Como Cadena;
6      nombre = "Carlos";
7      hijos = 0;
8      sueldo = 12345.67;
9      edad = 25;
10     Escribir sueldo;
11     Escribir nombre;
12     Escribir edad;
13     Escribir hijos;
14 FinAlgoritmo
```

Código 8: Algoritmo que carga y muestra variables.

Observarás que la salida corresponde con lo esperado: la primera línea escribe en la pantalla el contenido de la variable **sueldo**, que es el número real **12345.67**, y así sucesivamente.

En el ejemplo he puesto a propósito la definición, la asignación y la muestra de las variables en cualquier orden. Lo que sí es importante, reitero, es que primero la variable debe ser definida indicando su tipo de dato, luego se debe establecer un valor y a partir de allí estarías en condiciones de mostrarla.

El haber asignado un valor a una variable no significa que solo te quede como única opción mostrarla. Las variables se llaman así porque pueden cambiar su valor, si es que lo decidís, a lo largo del código.

Tras usar el operador de asignación, ahora nuestra memoria ha quedado así:

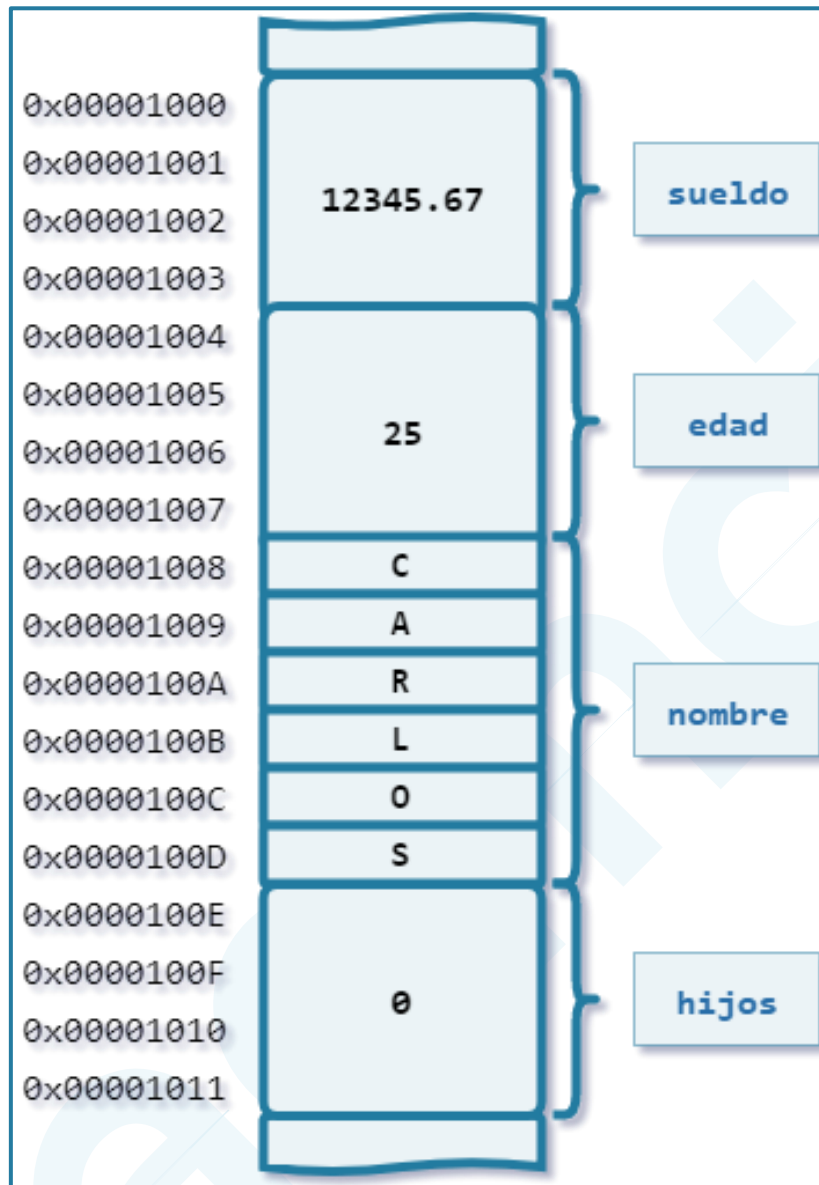


Ilustración 14: Simulación de memoria con variables inicializadas.

Asignar a una variable un valor que no se corresponde con el tipo de dato definido para ella, se producirá un error en tiempo de ejecución.

Es importante que asimiles que las asignaciones se leen de **derecha a izquierda**. El valor de la expresión o variable que está a la **derecha** del operador **=** se copia y se guarda en la variable que está a la **izquierda** del operador **=**. Si te acostumbras a esto desde temprano, no tendrás problema más adelante en comprender cómo actualizar variables según su resultado anterior.

Probá el siguiente programa:

```
1  Algoritmo modificacion_variable
2      Definir nombre Como Cadena;
3      Nombre = "Pepe";
4      Escribir nombre;
5      Nombre = "Luis";
6      Escribir nombre;
7  FinAlgoritmo
```

Código 9: Algoritmo que modifica el valor de una variable.

Tal como esperabas, la primera línea muestra **"Pepe"** y la segunda muestra **"Luis"**. Eso es suficiente para entender que la variable **nombre** ha cambiado su valor durante el transcurso del programa. Una variable **se define solo una vez al principio, sin importar si su valor es constante o cambia todo el tiempo**.

Por último, hay un caso que me gustaría mostrarte que puede traer confusión al principio:

```
1  Algoritmo diferencia_variable_cadena
2      Definir nombre Como Cadena;
3      nombre = "Pepe";
4      Escribir nombre;
5      Escribir "nombre";
6  FinAlgoritmo
```

Código 10: Algoritmo que diferencia entre un identificador de variable y una cadena.

Es importante que distingas que no es lo mismo **nombre** que **"nombre"**. Más allá del color, incentivando la diferencia, el primero es el identificador de una variable y el segundo es una cadena de caracteres. Por ello, la primera línea escribe el contenido de la variable **nombre** que es **"Pepe"** mientras que la segunda escribe literalmente la cadena **"nombre"**.

Actualizar una variable

Muchas veces te verás en la necesidad de establecer el valor de una variable según su resultado actual. Eso se llama actualizar una variable. Es muy usado sobre todo cuando veas flujo de repetición, en los que necesitarás contadores y acumuladores.

El siguiente programa le pide un valor al usuario, lo muestra y luego vuelve a mostrarlo, pero con su valor multiplicado por **2**. Si bien los cálculos pueden hacerse

directamente en la salida, la idea es que veas como se reutiliza la misma variable, actualizando su valor por el mismo pero duplicado:

```
1  Algoritmo actualizacion_variable
2      Definir num Como Entero;
3      Escribir "Ingresá un número:";
4      Leer num;
5      Escribir "El número ingresado es ",num;
6      num = num * 2;
7      Escribir "El número duplicado es ",num;
8  FinAlgoritmo
```

Código 11: Algoritmo que actualiza una variable.

La primera vez, el programa escribe el número tal cual lo ingresó el usuario. Pero luego la variable sufre una actualización.

La clave nuevamente es **leer la asignación de derecha a izquierda**: `num` tiene un valor entero introducido por el usuario que es multiplicado por `2`. El resultado de tal operación se asigna a `num`, pisando el valor anterior. Ahora `num` vale el doble de lo que valía antes. ¿Te das cuenta como se actualizó la variable?

Hasta aquí, has visto cómo establecer valores a las variables mediante el operador de asignación. Pero hay una segunda manera, que además hará que nuestros programas sean más dinámicos: **valores ingresados por el usuario**.

Instrucción de entrada

Ha llegado la hora de que tus programas sean realmente más dinámicos y de utilidad práctica. Hasta aquí lo que hacías no era muy sofisticado. Solo podías mostrar palabras o frases y hacer alguna operación matemática. No existe aplicación alguna donde un usuario no interactúe con ella.

Lo primero a conocer es la instrucción que permite recibir datos mediante el usuario.

La sintaxis es la siguiente:

Leer <variable>;

- **Leer** se tipea literalmente, indica que queremos obtener un dato desde la entrada estándar, que es el teclado.
- La palabra **<variable>** indica que en ese lugar debe ir el identificador de una variable definida, que será quien guarde el valor tipeado por el usuario.
- El **;** es obligatorio e indica que finalizó la instrucción.

Cada vez que el intérprete pasa por una instrucción de lectura, se detiene el flujo del programa y se pasa el turno al usuario. Hasta que éste no escriba un valor y a continuación tipee **Enter**, la ejecución no continuará.

El siguiente ejemplo sencillo permite que el usuario ingrese dos números para que la computadora devuelva el resultado de su suma:

```
1  Algoritmo sumador
2      Definir numero1 Como Entero;
3      Definir numero2 Como Entero;
4      Escribir "SUMADOR";
5      Escribir "Ingrese el primer número entero:";
6      Leer numero1;
7      Escribir "Ingrese el segundo número entero:";
8      Leer numero2;
9      Escribir "El resultado es";
10     Escribir numero1 + numero2;
11  FinAlgoritmo
```

Código 12: Algoritmo que suma dos enteros provistos por el usuario.

Justo antes de una instrucción de lectura, recomiendo que le escribas por pantalla al usuario qué es lo que esperas que introduzca, a fin de que el programa sea lo más amigable posible.

El programa comienza imprimiendo las primeras dos líneas y se detiene esperando que se ingrese un número entero. Es aquí donde se te traslada la responsabilidad de que ingrese cualquier entero y luego presiones **Enter**. El programa reanuda su ejecución, imprime la siguiente línea y vuelve a detenerse. Debés ingresar el segundo número y presionar **Enter**. Por último, se imprime la última cadena y se escribe debajo el resultado de la expresión, que representa la suma de los números que ingresaste. ¡Felicidades, acabás de crear tu primera sumadora de enteros!

Si el usuario ingresa un valor que no se corresponde con el tipo de dato definido para la variable, se producirá un error en tiempo de ejecución.

Si hay algo que puliría en lo que acabás de hacer, es que primero sale una cadena que dice **"El resultado es"** y en una línea aparte el resultado. ¿No hubiese sido mejor que se visualice todo junto en una sola línea? Entonces, **necesitamos alguna forma de juntar cadenas con variables**.

Operador de concatenación

Para hacer más atractivas las salidas, se utiliza el operador de concatenación, que es la coma. Este permite unir cadenas con otras o con contenidos de variables.

Es indistinto dejarla pegada o con espacios entre las partes. Donde sí hay diferencia es dentro de las comillas que delimitan una cadena.

Si reformulamos el sumador aplicando el operador de concatenación a la última salida, obtenemos el siguiente código:

```
1  Algoritmo sumador_salida_concatenada
2      Definir numero1 Como Entero;
3      Definir numero2 Como Entero;
4      Escribir "SUMADOR";
5      Escribir "Ingrese el primer número entero:";
6      Leer numero1;
7      Escribir "Ingrese el segundo número entero:";
8      Leer numero2;
9      Escribir "El resultado es " , (numero1 + numero2);
10 FinAlgoritmo
```

Código 13: Algoritmo que suma dos enteros y utiliza el operador de concatenación.

Ahora el resultado sale en una sola línea de manera más amigable.

El uso de paréntesis en la expresión no es obligatorio, pero los incluí para que el código quede más legible. Quiero que notes que la palabra **"es"** y el resultado de la suma salen espaciados gracias a que coloqué un espacio justo antes de cerrar la comilla doble. Ese espacio es un carácter más, por lo que sale literalmente, haciendo que luego el número no quede pegado a la cadena. Los espacios que dejé al principio y al final del operador de concatenación tienen motivos de legibilidad y no tienen efecto alguno en la ejecución final.

Uso de funciones

Una función es un trozo de código que resuelve un pequeño problema en particular. Digamos que no es necesario reinventar la rueda. La misma ya ha sido inventada y solo requerimos saber usarla.

No es el momento para explicarte como crear una función cuando recién estás conociendo los primeros conceptos de la programación. Pero sí es conveniente que sepas invocar una función para ayudarte a procesar datos. Supongamos que tenés que calcular la raíz cuadrada de un número positivo. No hay un operador que lo haga y

además, todavía no has aprendido las herramientas que te permitirían codificar un algoritmo que lo calcule.

Con la ayuda de una función, podrías simplemente invocarla con el dato a resolver y ésta te devolvería el valor calculado. El dato que se le pasa a una función se denomina **argumento**. Hay funciones que no requieren argumentos, o solo uno, o varios, además de devolver un tipo de dato diferente, según su propósito.

PSelnt cuenta con una biblioteca de funciones básicas, pero antes, tenés que comprender cómo se invoca una función.

La sintaxis es la siguiente:

<función>(<argumento1>,<argumento2>,...,<argumentoN>)

- **<función>** se reemplaza por el nombre de la función a invocar. Enseguida verás algunos ejemplos.
- Luego del nombre de la función, deben abrirse paréntesis. Dentro de ellos, se introducen los argumentos, separados por comas. Si una función no requiere argumentos, de todas maneras, hay que dejar los paréntesis (quedarán vacíos).
- **<argumento>** se reemplaza por el valor que la función requiera. Puede ser un número o una cadena. Todo depende de la función en cuestión.

Te voy a mostrar un ejemplo. El siguiente programa le pide al usuario un número positivo para que la máquina calcule su raíz cuadrada. **Si el número ingresado no es positivo, el programa fallará.** Recordá que matemáticamente, no se puede resolver la raíz cuadrada de un número negativo.

Para calcular la raíz cuadrada, voy a invocar a la función **rc()**, que requiere como único argumento un número positivo:

```
1  Algoritmo funciones
2      Definir num Como Real;
3      Definir resultado Como Real;
4      Escribir "Ingresá un número positivo";
5      Leer num;
6      resultado = rc(num);
7      Escribir "La raíz cuadrada de ", num , " es " , resultado;
8  FinAlgoritmo
```

Código 14: Algoritmo que calcula la raíz cuadrada de un número usando funciones.

Suponiendo que el usuario ingrese el **25**, fijate como invoco a la función **rc()** pasándole como argumento la variable **num**, que guarda este valor. La función **rc()**

toma el argumento, hace los cálculos y devuelve el resultado, que es **5** y a continuación se asigna a la variable **resultado** mediante el operador de asignación. Lo siguiente es mostrar el resultado por pantalla.

PSelnt cuenta con 20 funciones ya armadas, que podés desplegar desde el botón que está a la izquierda de la ventana llamado **Operadores y Funciones**.

A continuación, la lista de funciones junto a su descripción y un ejemplo:

Funciones para números

- **abs(<exp_numérica>)**

Devuelve el valor absoluto del argumento.

abs(-8.7) devuelve **8.7**

- **acos(<exp_numérica>)**

Devuelve el arco coseno del argumento.

acos(1) devuelve **0**

- **asen(<exp_numérica>)**

Devuelve el arco seno del argumento.

asen(0) devuelve **0**

- **atan(<exp_numérica>)**

Devuelve el arco tangente del argumento.

atan(0) devuelve **0**

- **azar(<entero_positivo>)**

Devuelve un número entero al azar entre 0 y el argumento menos uno.

azar(3) devuelve un número entero entre **0** y **2**

- **cos(<exp_numérica_en_radianes>)**

Devuelve el coseno del argumento.

cos(0) devuelve **1**

- **exp(<exp_numérica>)**

Devuelve el resultado de e exponenciado al argumento.

exp(1) devuelve **2.7181828185**

- `ln(<exp_numérica_positiva>)`

Devuelve el logaritmo natural del argumento.

`ln(2.7181828185)` devuelve `1`

- `rc(<exp_numérica_no_negativa>)`

Devuelve la raíz cuadrada del argumento.

`rc(81)` devuelve `9`

- `redon(<exp_numérica>)`

Devuelve el valor redondeado del argumento.

`redon(-8.7)` devuelve `-9`

- `sen(<exp_numérica>)`

Devuelve el seno del argumento.

`sen(0)` devuelve `0`

- `tan(<exp_numérica>)`

Devuelve la tangente del argumento.

`tan(0)` devuelve `0`

- `trunc(<exp_numérica>)`

Devuelve el valor truncado del argumento.

`abs(-8.7)` devuelve `-8.`

Funciones para cadenas

- `concatenar(<cadena1>, <cadena2>)`

Devuelve una cadena con los argumentos concatenados.

`concatenar("Hola", "Mundo")` devuelve `"HolaMundo"`

- `convertirANumero(<cadena>)`

Devuelve un número que coincide con la representación numérica del argumento.

`convertirANumero("257")` devuelve `257`

- `convertirATexto(<exp_numérica>)`

Devuelve una cadena que representa el número.

`convertirATexto(257)` devuelve `"257"`

- `longitud(<cadena>)`

Devuelve la longitud del argumento.

`longitud("Hola")` devuelve `4`

- `mayusculas(<cadena>)`

Devuelve una cadena que representa el argumento en mayúsculas.

`mayusculas("Hola")` devuelve `"HOLA"`

- `minusculas(<cadena>)`

Devuelve una cadena que representa el argumento en minúsculas.

`minusculas("Hola")` devuelve `"hola"`

- `subCadena(<cadena>, <entero1>, <entero2>)`

Devuelve una cadena con los caracteres del primer argumento, desde la posición del primer argumento hasta la posición del segundo argumento, ambos incluidos.

`subCadena("Pizarrón", 2, 4)` devuelve `"zar"`

(Los caracteres se cuentan a partir del cero).

Comentarios

Una muy buena práctica a la hora de programar es comentar el código. Esto significa añadir notas que ayuden a entender alguna instrucción compleja o para describir tareas pendientes. No son tenidos en cuenta por el intérprete, solo sirven para el programador.

Hay dos tipos de comentarios: **de línea** y **de bloque**. PSeInt solo cuenta con el primero.

Un comentario de línea se inserta con una doble barra, sin espacios, de esta manera:

`/**`. Verás que el texto a continuación de allí se pone en gris, indicando que no será tenido en cuenta por el intérprete. Desde un `/**` en adelante, se trata de un comentario.

A continuación, te dejo como ejemplo el sumador anterior con algunos comentarios sencillos:

1	<code>Algoritmo comentarios</code>
2	<code>Definir numero1 Como Entero; // Declaración de variable</code>

```
3 Definir numero2 Como Entero; // Declaración de variable
4 Escribir "SUMADOR";
5 Escribir "Ingrese el primer número entero:";
6 Leer numero1; // Capturo el primer número
7 Escribir "Ingrese el segundo número entero:";
8 Leer numero2; // Capturo el segundo número
9 // Puedo poner un comentario que ocupe una línea completa
10 Escribir "El resultado es " , (numero1 + numero2);
11 FinAlgoritmo
```

Código 15: Algoritmo con comentarios

Aunque te parezca redundante comentar algo que ya acabas de hacer, es una gran ayuda para quien vea tu código. Tené en cuenta que en la creación de software a gran escala trabaja un equipo de cientos de programadores. Inclusive los comentarios son valiosos para tu "yo del futuro", es decir, si acabás de programar algo complejo que te llevó un buen rato deducir, comentarlo hará que cuando lo releas más adelante, te sea más fácil interpretar lo que hiciste.

Integrando los conceptos

Para darle un cierre a este capítulo, te presento un programa que reúne todos los temas vistos hasta aquí. Probalo:

```
1 Algoritmo saludador
2 Definir nombre Como Cadena;
3 Definir anioActual Como Entero; // No se puede usar ñ
4 Definir anioDeNacimiento Como Entero; // No se puede usar ñ
5 Definir edad Como Entero;
6 Escribir "Ingresá tu nombre:";
7 Leer nombre;
8 Escribir "Hola " , nombre , ", ¡un placer!";
9 Escribir "Tu nombre tiene " , longitud(nombre) , " letras.";
10 Escribir "¿Podrías decirme en qué año estamos actualmente?";
11 Leer anioActual;
12 Escribir "Bien, ¿vos en que año naciste?";
13 Leer anioDeNacimiento;
```

```

14     edad = anioActual - anioDeNacimiento;
15     Escribir "Tenés " , edad , " años, si es que ya los cumpliste.";
16     FinAlgoritmo

```

Código 16: Ejemplo integrador de los conceptos de flujo secuencial.

Ejecución paso a paso

Una de las mejores herramientas con las que cuenta PSeInt es la **ejecución paso a paso**, que podés activar mediante el botón . El programa ejecuta la aplicación, pero a una velocidad que permite hacer un seguimiento de lo que hace cada instrucción. A la derecha podés encontrar un panel con opciones como una barra deslizable para ajustar la velocidad de ejecución o botones para controlar el flujo del programa manualmente.

Generar diagrama de flujo

PSeInt es capaz de **generar un diagrama de flujo** a partir de un pseudocódigo, a través del botón .

A través del botón , podés guardar el diagrama de flujo que estás visualizando como imagen.

Además de seguir el estándar de figuras que te mostré en la página **13**, el programa incorpora colores que ayudan a identificar qué instrucciones corresponden con delimitadores, entradas, procesos y salidas.

Forma	Descripción
	Delimitador. Representa el comienzo o la finalización de la secuencia de instrucciones.
	Entrada. Representa la adquisición de datos por medio del teclado. Aparece por cada instrucción Leer en el código.
	Proceso. Representa una instrucción.
	Salida. Representa la visualización de los resultados por medio de la consola. Aparece por cada instrucción Escribir en el código.

	Condición. Representa un interrogante cuya evaluación debe dar únicamente VERDADERO o FALSO .
	Línea de flujo. Representa la dirección de la secuencia de instrucciones.

El diagrama de flujo correspondiente al ejemplo integrador es el siguiente:

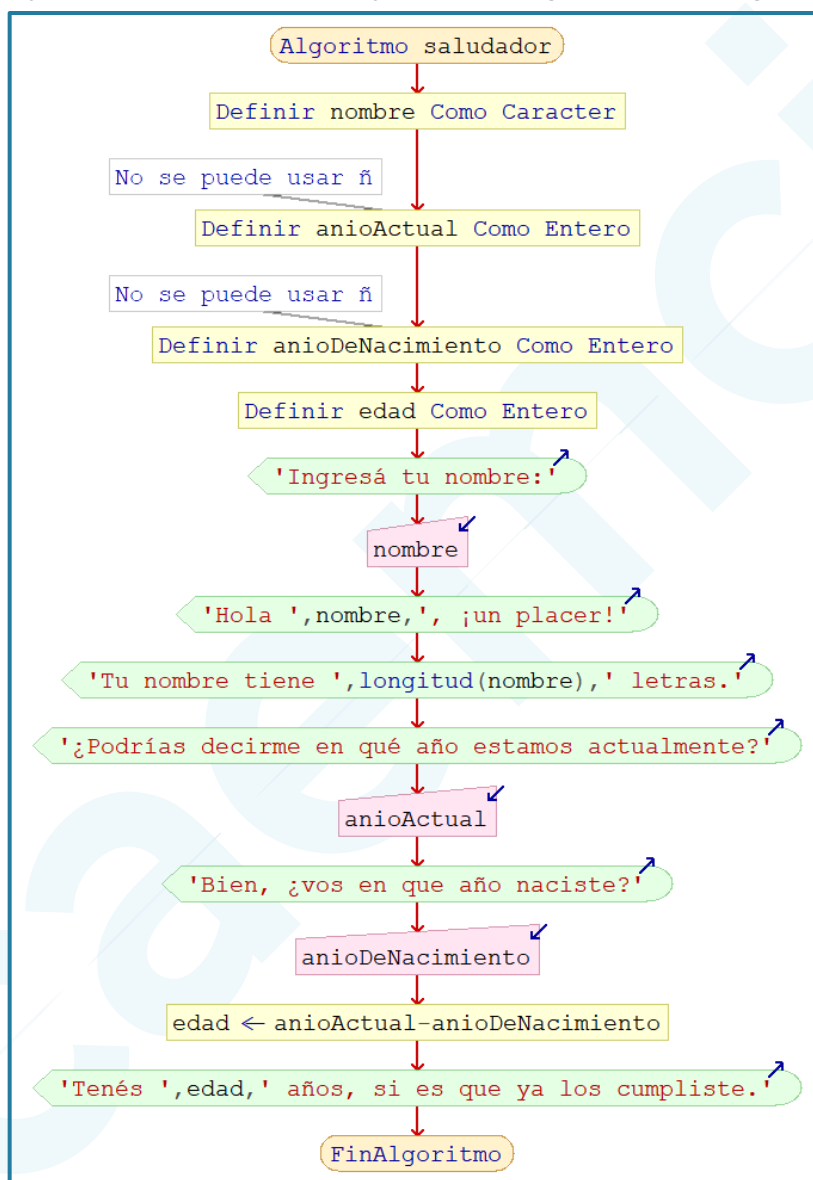


Ilustración 15: Ejemplo integrador de los conceptos de flujo secuencial.

Hasta acá todos los programas realizados tienen algo en común: las instrucciones se realizan de arriba hacia abajo, paso a paso, cual receta de cocina.

A continuación, verás como alterar este flujo secuencial.

Flujo de selección

A la hora de resolver problemas más complejos, es necesario poder contar con alguna manera de poder discriminar entre la ejecución de ciertas instrucciones u otras en base a cierto criterio. Esto abre un nuevo abanico de posibilidades para la resolución de problemas, ya que la máquina será capaz de tomar uno u otro camino dependiendo de una condición planteada de antemano por el programador.

Si los diagramas de flujo anteriores tenían la forma de una línea recta, de principio a fin, instrucción por instrucción, ahora nuestros programas podrán bifurcarse una o más veces.

Lo primero que hay que conocer, son conceptos básicos de la **lógica proposicional**, una rama de la lógica matemática, que se encarga de estudiar afirmaciones o proposiciones y su relación entre ellas.

Una **proposición** es un enunciado mínimo al que se le puede atribuir un **valor de verdad**. La proposición puede ser **verdadera** (lo que enuncia es correcto) o **falsa** (lo que enuncia no es correcto).

Es importante que comprendas que solo existen dos valores de verdad posibles: **VERDADERO** o **FALSO**. No existen los "no se sabe", los "quizás" ni los "más o menos". Además, una proposición no puede tener los dos valores al mismo tiempo. O es verdadera, o es falsa.

A continuación, te muestro algunas proposiciones, junto a su valor de verdad:

Proposición	Valor de verdad
"El Sol es una estrella"	VERDADERO
"La Tierra es plana"	FALSO
"Tres por cuatro, es doce"	VERDADERO
"Einstein era alemán"	VERDADERO
"Argentina está en Europa"	FALSO
"Sevilla es la capital de España"	FALSO

Expresiones booleanas

Para poder establecer condiciones y que la máquina las evalúe para saber qué camino tomar, se debe hacer uso de **expresiones booleanas**. Hablé de ellas en la página 29.

Así como has visto en la página 30 que una expresión aritmética, por muy compleja que sea, siempre resulta en un número, **una expresión booleana resulta en un valor lógico (VERDADERO o FALSO).**

Si para una expresión aritmética, es necesario relacionar números con operadores aritméticos, para obtener una expresión booleana debemos relacionar expresiones con operadores relacionales y lógicos.

Operadores relacionales

Son aquellos que permiten comparar expresiones. Si la evaluación es correcta, la máquina retorna **VERDADERO**, de lo contrario, retorna **FALSO**.

Operador	Nombre	Ejemplo	Resultado
<	Menor que	4 < 5	V
<=	Menor o igual que	5 <= 5	V
>	Mayor que	4 > 5	F
>=	Mayor o igual que	5 >= 5	V
==	Igual que	4 == 5	F
!=	Distinto que (para números)	4 != 5	V
<>	Distinto que (para cadenas)	"PEZ" <> "pez"	V

Es muy común al principio confundir el operador de asignación (=) con el de comparación (==). PSeInt puede dilucidar que, al poner, por ejemplo:

Escribir 2 = 2;

en realidad, quisiste comparar dos números entre sí, por lo que te devolverá un valor lógico. Pero en los lenguajes formales eso sería un error. Estarías queriendo asignar un 2 a un 2. No tiene sentido. Comenzá desde temprano a prestar atención y notar el contexto del código, para saber si se trata de una asignación (con =) o una comparación (con ==).

Los operadores relacionales tienen menor prioridad que los aritméticos, esto significa que, por ejemplo, la siguiente expresión: $5 + 4 < 3 - 2$, será evaluada primero en términos de operadores aritméticos, quedando la siguiente expresión: $9 < 1$, donde se terminan de evaluar los operadores relacionales que hubiere (en este caso, solo uno), dando como resultado **FALSO** (9 no es menor que 1).

Para comprobar como PSeInt evalúa las expresiones booleanas, te invito a que pruebes el siguiente programa, que cuenta con unas cuantas de ellas, de menor a mayor complejidad. Tu tarea no tan solo es ver los resultados de la ejecución del programa, sino que evalúes las expresiones vos también de manera manual para comprobar que el resultado es el mismo.

```

1  Algoritmo expresiones_booleanas_1
2      Escribir 2 == 3; // Debería dar FALSO
3      Escribir 2 != 3; // Debería dar VERDADERO
4      Escribir 2 < 3; // Debería dar VERDADERO
5      Escribir 2 > 3; // Debería dar FALSO
6      Escribir 2 >= 3; // Debería dar FALSO
7      Escribir 2 <= 3; // Debería dar VERDADERO
8      Escribir 5 + 4 < 3 - 2; // Debería dar FALSO
9      Escribir 2 < 3 - 4 * 5; // Debería dar FALSO
10     Escribir 2 - 3 * 0 >= rc(3 + 1); // Debería dar VERDADERO
11 FinAlgoritmo

```

Código 17: Verificación de expresiones booleanas con operadores relacionales entre números.

No solo se pueden utilizar operadores relacionales para comparar expresiones numéricas, sino también caracteres y cadenas. El criterio se considera **según la tabla de caracteres de la página Ilustración 12: Tabla de caracteres ASCII.28**. Por ejemplo, el carácter 'A' es menor que 'B', ya que aparece antes en la tabla. La palabra "arena" es menor que la palabra "armadura", ya que "arena" aparece antes en el diccionario. También hay que ser cautos con la diferencia entre mayúsculas y minúsculas: la palabra "Zapato" es menor que la palabra "abeja", ya que la 'Z' en mayúsculas aparece antes que la 'a' en minúsculas.

Lo mejor es que ejecutes el siguiente programa y compruebes los resultados:

```

1  Algoritmo expresiones_booleanas_2
2      Escribir 'A' < 'B'; // Debería dar VERDADERO
3      Escribir "arena" < "armadura"; // Debería dar VERDADERO

```

```

4      Escribir "Zapato" < "abeja"; // Debería dar VERDADERO
5      Escribir "Borrador" < "Borra"; // Debería dar FALSO
6      Escribir "tecla" < "TECLA"; // Debería dar FALSO
7      FinAlgoritmo

```

Código 18: Verificación de expresiones booleanas con operadores relacionales entre cadenas.

Flujo de selección simple

La primera y más sencilla manera de alterar el flujo secuencial por defecto de cualquier programa estructurado es utilizar una estructura de selección simple.

Observá el siguiente diagrama:

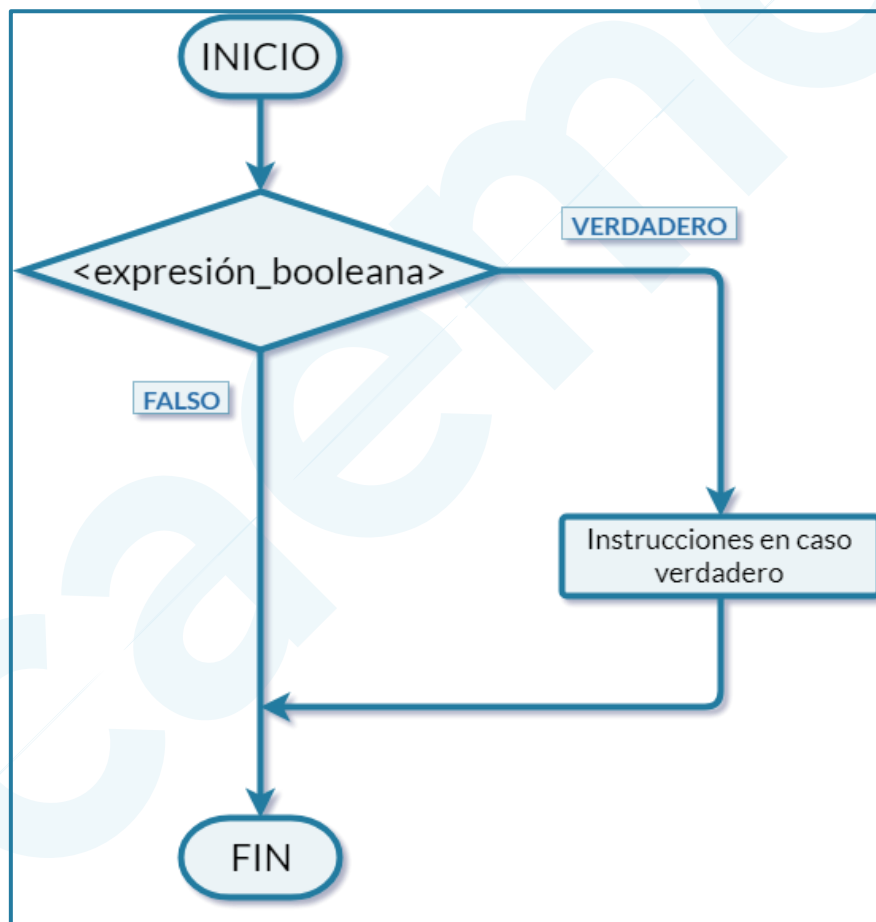


Ilustración 16: Flujo de selección simple.

En cierto momento, definido por el programador, la computadora evalúa una condición, es decir, una expresión booleana, que se representa mediante un rombo. Si la expresión devuelve un resultado **VERDADERO**, la computadora ejecuta las

instrucciones dentro de un bloque especial que luego retorna al flujo original, de lo contrario, el flujo continúa normalmente.

La sintaxis en pseudocódigo es la siguiente:

Si (<expresión_booleana>) **Entonces**

 <instrucciones>

FinSi

- **Si** se tipea literalmente, indica que vamos a iniciar un bloque de selección.
- <expresión_booleana> es la condición que la computadora evaluará. El hecho de estar entre paréntesis es opcional, aunque te recomiendo que lo hagas para mejorar la legibilidad y porque muchos lenguajes formales requieren que la condición esté obligatoriamente encerrada entre paréntesis.
- **Entonces** se tipea literalmente, indica que a continuación serán listadas las instrucciones que se ejecutarán en caso de que la condición sea **VERDADERO**.
- <instrucciones> son las instrucciones que se ejecutarán. Puede ser solo una, o varias, siguiendo los conceptos vistos hasta aquí.
- **FinSi** se tipea literalmente, indica que ha terminado el bloque de selección.

Un programa sencillo que facilita la comprensión podría ser el siguiente: pedirle al usuario que ingrese su edad. En caso de que tenga menos de 18 años, mostrarle un mensaje que diga **"No podés ingresar"**. Sea cual fuere la edad, el programa termina diciendo **"Suerte"**.

El código sería el siguiente:

```
1  Algoritmo seleccion_simple
2      Definir edad Como Entero;
3      Escribir "Ingresá tu edad:";
4      Leer edad;
5      Si (edad < 18) Entonces
6          Escribir "No podés ingresar"; // Se escribe según la edad
7      FinSi
8      Escribir "Suerte"; // Se escribe siempre
9  FinAlgoritmo
```

Código 19: Algoritmo que valida un pase según la edad.

Si probás el programa anterior con un valor igual o mayor a **18**, por ejemplo, **30**, la computadora evaluará la condición como **FALSO**, por lo que las instrucciones que están dentro del bloque **Si...FinSi** no serán ejecutadas. La ejecución continúa justo después del bloque **Si...FinSi**, encontrándose con la escritura de la palabra **"Suerte"** y dando por finalizada la ejecución.

Si probás el programa anterior con un valor menor a **18**, por ejemplo, **12**, la computadora evaluará la condición como **VERDADERO**, por lo que las instrucciones que están dentro del bloque **Si...FinSi** serán ejecutadas. La ejecución continúa justo después del bloque **Si...FinSi**, encontrándose con la escritura de la palabra **"Suerte"** y dando por finalizada la ejecución.

Es importante que notes que la palabra **"Suerte"** se escribe siempre, ya que al estar fuera del bloque **Si...FinSi**, su ejecución no se encuentra condicionada.

El diagrama de flujo que genera PSeInt es el siguiente:

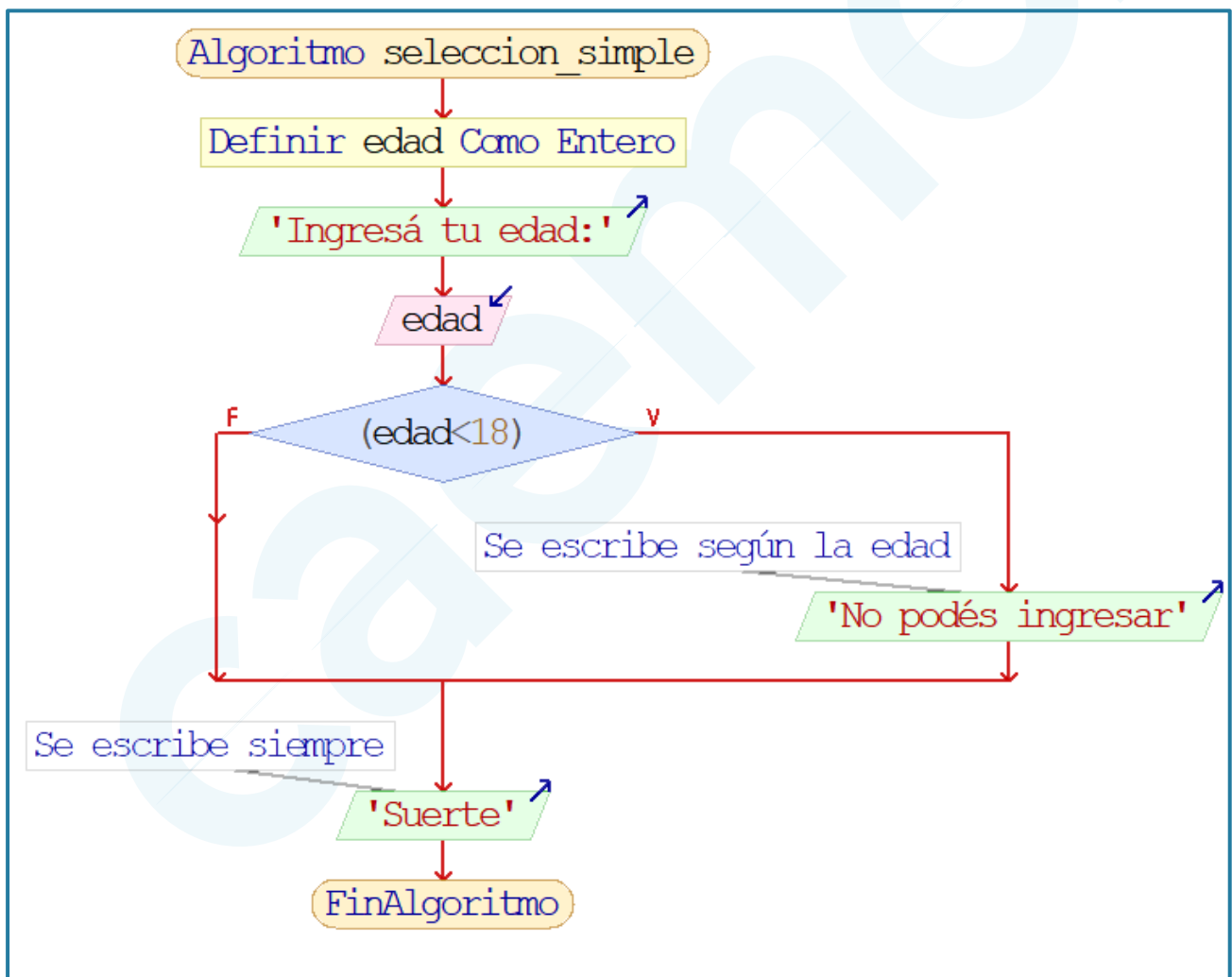


Ilustración 17: Algoritmo que valida un pase según la edad.

Una buena práctica que mejora la legibilidad del programa es darle sangría a las instrucciones que están dentro de un bloque **Si...FinSi**. Tal acción permite un

código ordenado que facilita mucho la depuración a medida que el mismo se torna más complejo.

Para dejar una sangría en una instrucción, basta con poner el cursor justo delante del primer carácter y tocar la **tecla tabulador (TAB)**. A este proceso se lo conoce como "**indentar**", aunque el término no es una palabra reconocida del español, sino un anglicismo de la palabra inglesa "**indentation**".

Flujo de selección doble

La estructura anterior permite realizar una serie de instrucciones en caso de que una condición sea **VERDADERO**, pero no especifica nada en caso de que sea **FALSO**. De hecho, si la condición resulta **FALSO**, el programa continúa su ejecución como si el bloque **Si...FinSi** no existiera.

Para contemplar y hacer una serie de instrucciones específicas en caso de que una condición resulte **FALSO**, sin dejar de lado lo que se hacía cuando resultaba **VERDADERO**, es necesario utilizar una estructura de selección doble, según el siguiente diagrama:

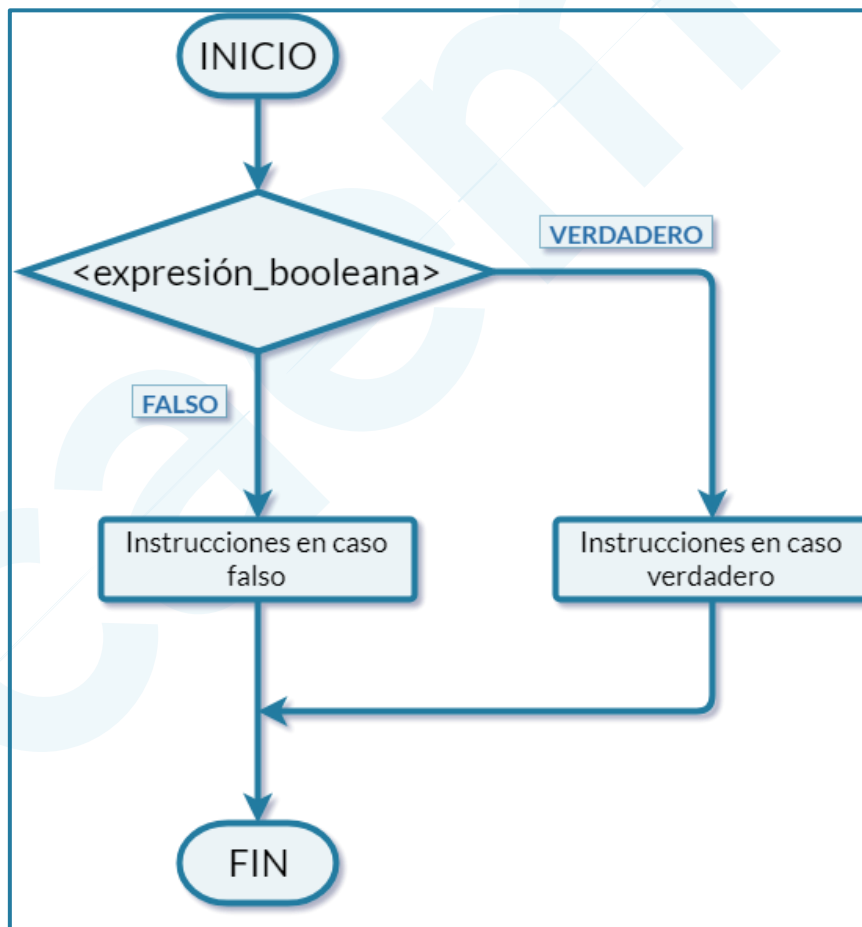


Ilustración 18: Flujo de selección doble.

Te invito a que te tomes el trabajo de comparar el diagrama de flujo de selección doble con el de selección simple. La única diferencia es que ahora sí podemos tomar acción en caso de que la condición resulte **FALSO**.

La sintaxis en pseudocódigo es la siguiente:

```
Si (<expresión_booleana>) Entonces
    <instrucciones_caso_VERDADERO>
Sino
    <instrucciones_caso_FALSO>
FinSi
```

- **Si** se tipea literalmente, indica que vamos a iniciar un bloque de selección.
- **<expresión_booleana>** es la condición que la computadora evaluará. El hecho de estar entre paréntesis es opcional, aunque te recomiendo que lo hagas para mejorar la legibilidad y porque muchos lenguajes formales requieren que la condición esté obligatoriamente encerrada entre paréntesis.
- **Entonces** se tipea literalmente, indica que a continuación serán listadas las instrucciones que se ejecutarán en caso de que la condición sea **VERDADERO**.
- **<instrucciones_caso_VERDADERO>** son las instrucciones que se ejecutarán, si es que la condición resulta **VERDADERO**. Puede ser solo una, o varias, siguiendo los conceptos vistos hasta aquí.
- **Sino** se tipea literalmente, indica que a continuación serán listadas las instrucciones que se ejecutarán en caso de que la condición sea **FALSO**.
- **<instrucciones_caso_FALSO>** son las instrucciones que se ejecutarán, si es que la condición resulta **FALSO**. Puede ser solo una, o varias, siguiendo los conceptos vistos hasta aquí.
- **FinSi** se tipea literalmente, indica que ha terminado el bloque de selección.

Continuando con el programa sencillo de ejemplo anterior, vamos a pedirle al usuario que ingrese su edad. En caso de que tenga menos de 18 años, mostrarle un mensaje que diga "**No podés ingresar**", en caso contrario, dirá "**Te damos la bienvenida, ¡pasá!**". Sea cual fuere la edad, el programa termina diciendo "**Suerte**".

El código sería el siguiente:

1	Algoritmo seleccion_doble
2	Definir edad Como Entero;

```

3      Escribir "Ingresá tu edad:";
4      Leer edad;
5      Si (edad < 18) Entonces
6          Escribir "No podés ingresar"; // Si es VERDADERO
7      Sino
8          Escribir "Te damos la bienvenida, ¡pasá!";
9      FinSi
10     Escribir "Suerte"; // Se escribe siempre
11     FinAlgoritmo

```

Código 20: Algoritmo que valida o no un pase según la edad.

El diagrama de flujo que genera PSeInt es el siguiente:

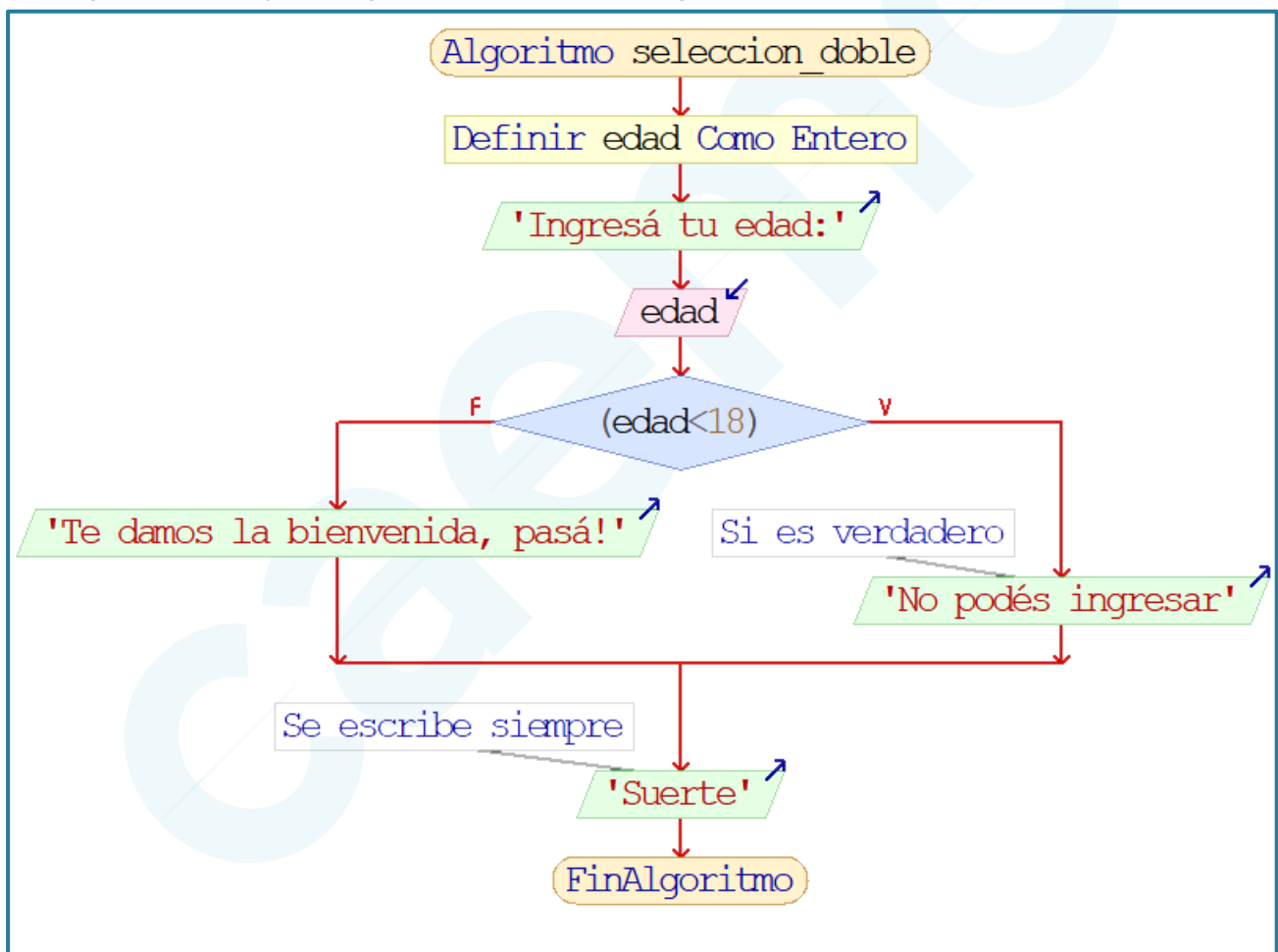


Ilustración 19: Algoritmo que valida o no un pase según la edad.

Si probás el programa anterior con un valor igual o mayor a 18, por ejemplo, 30, la computadora evaluará la condición como **FALSO**, por lo que se ejecutarán las instrucciones que estén justo después del **Sino**. La ejecución continúa justo después

del bloque **Si...FinSi**, encontrándose con la escritura de la palabra **"Suerte"** y dando por finalizada la ejecución.

Si probás el programa anterior con un valor menor a **18**, por ejemplo, **12**, la computadora evaluará la condición como **VERDADERO**, por lo que se ejecutarán las instrucciones que estén justo después del **Si**, concluyendo justo antes del **Sino**. La ejecución continúa justo después del bloque **Si...FinSi**, encontrándose con la escritura de la palabra **"Suerte"** y dando por finalizada la ejecución.

Es importante que notes que la palabra **"Suerte"** se escribe siempre, ya que al estar fuera del bloque **Si...FinSi**, su ejecución no se encuentra condicionada.

Fijate que continúo "indentando" a las instrucciones dentro de un bloque **Si-Sino**, para mejorar la legibilidad.

Para poder realizar programas más complejos, que contemplen más posibilidades (o caminos), basta con poder anidar estructuras de selección, cuestión que veremos en el siguiente apartado.

Anidamiento de estructuras de selección

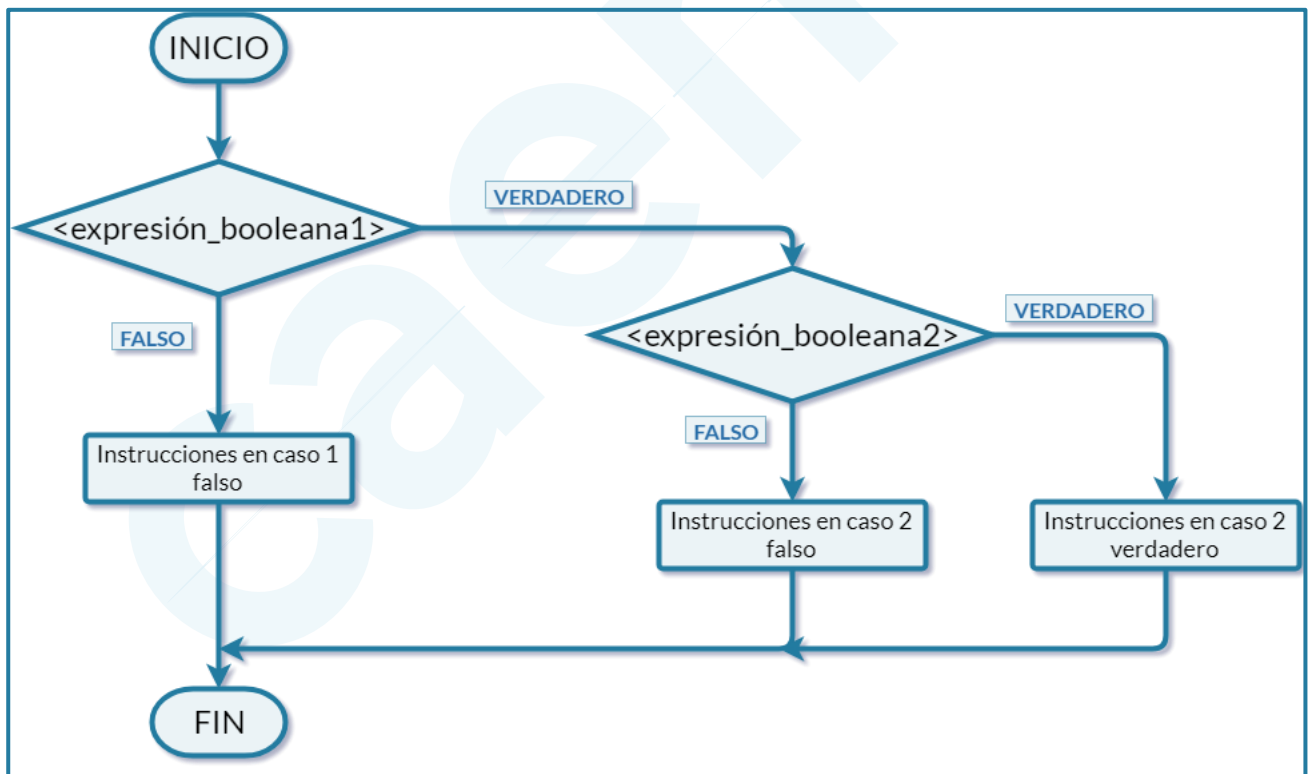


Ilustración 20: Anidamiento de estructuras de selección.

Para comprender este tema, la idea es realizar el siguiente algoritmo: el usuario ingresa un número y la computadora indica si se trata de un número positivo, negativo o el cero.

Habrás visto que la máquina solo puede resolver expresiones booleanas, las cuales solos tienen dos resultados posibles: **VERDADERO** o **FALSO**. Como nuestro programa evidentemente contempla tres posibles acciones, debemos hacer uso de **anidamiento de estructuras de selección**, obteniendo un flujo como el siguiente:

Un ejemplo puede ser el siguiente: si el número ingresado es mayor que 0, se trata sin discusión de un **número positivo**, por lo tanto, suponiendo que alojo el número en una variable llamada **num**, la expresión **num > 0** resultaría **VERDADERO**. En caso **FALSO**, no puedo asegurar que se trate de un número negativo, pues pudo haber sido el 0. En ese caso, se tiene que volver a emplear una estructura de selección, cuya condición puede ser **num < 0**. Si la expresión anterior resulta **VERDADERO**, puedo asegurar que se trata de un **número negativo**, en caso contrario, puedo asegurar que se trata del 0.

Una imagen vale más que mil palabras. Observá cómo queda el diagrama de flujo:

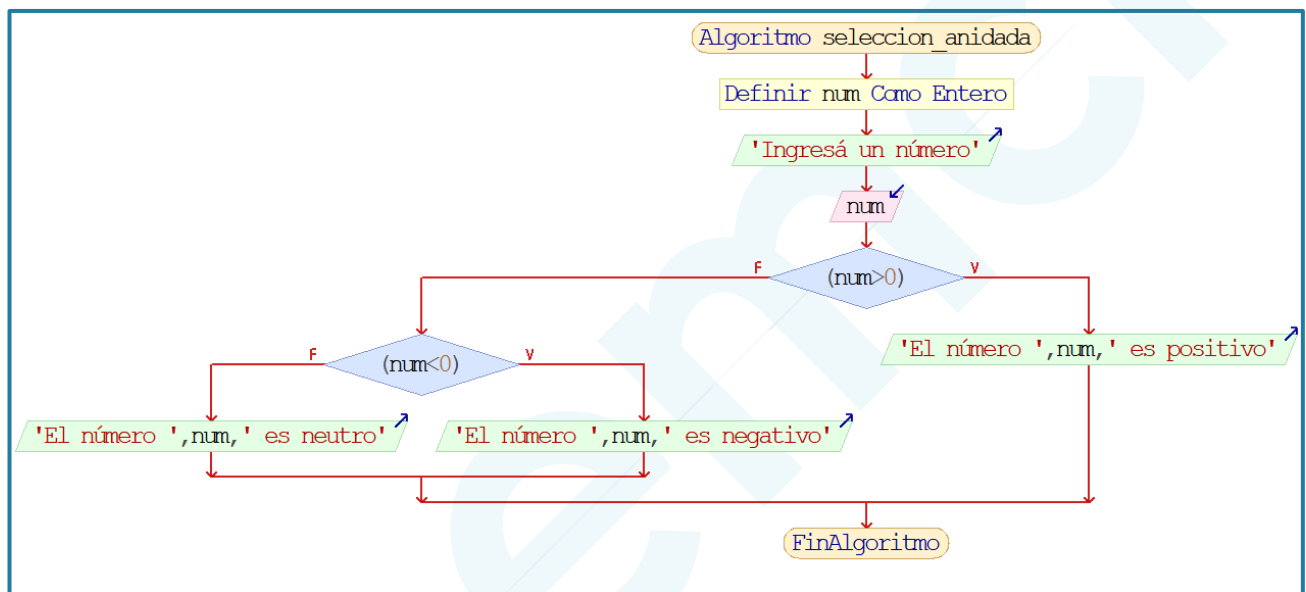


Ilustración 21: Algoritmo que reconoce el signo de un número.

Traduciendo el diagrama a código, queda lo siguiente:

```

1  Algoritmo seleccion_anidada
2      Definir num Como Entero;
3      Escribir "Ingresá un número";
4      Leer num;
5      Si (num > 0) Entonces
6          Escribir "El número " , num , " es positivo";
7      SiNo
8          Si (num < 0) Entonces
9              Escribir "El número " , num , " es negativo";
  
```

```

10      SiNo
11          Escribir "El número " , num , " es neutro";
12      FinSi
13  FinSi
14 FinAlgoritmo

```

Código 21: Algoritmo que reconoce el signo de un número.

Por supuesto que el ejemplo anterior puede hacerse de muchas formas. Por ejemplo, puedo primero poner una condición `num == 0`. Si el resultado es **VERDADERO**, el número es el **0**, de lo contrario, debo preguntar por alguna de las dos posibilidades: si es mayor, o si es menor. Elijo la segunda. Si resulta **VERDADERO**, el número es **negativo**, de lo contrario, es **positivo**.

Voy a poner otro ejemplo un poco más complejo. El usuario debe ingresar un valor numérico entero positivo, que representa la hora del día. Los números pueden estar comprendidos en el rango de **0** a **23**, ambos inclusive. Cualquier valor fuera de ese rango debe mostrarle al usuario en la pantalla un mensaje de error.

Suponiendo que el valor ingresado está comprendido en este rango, la computadora informa a qué momento del día pertenece, según el siguiente criterio:

- 0 a 11: Mañana
- 12: Mediodía
- 13 a 19: Tarde
- 20 a 23: Noche

Para poder realizar este ejercicio, lo primero que yo haría es validar que el usuario haya ingresado un valor entre **0** y **23**. Puedo hacerlo de la siguiente manera:

```

1  Algoritmo horarios1
2      Definir hora Como Entero;
3      Escribir "Ingresá una hora del día";
4      Leer hora;
5      Si (hora < 0) Entonces
6          Escribir "La hora ",hora," no existe";
7      SiNo
8          Si (hora > 23) Entonces
9              Escribir "La hora ",hora," no existe";

```

10	SiNo
11	Escribir "La hora ",hora," es correcta";
12	FinSi
13	FinSi
14	FinAlgoritmo

Código 22: Primera parte del algoritmo reconocedor del momento del día.

Quedando un diagrama de flujo como este:

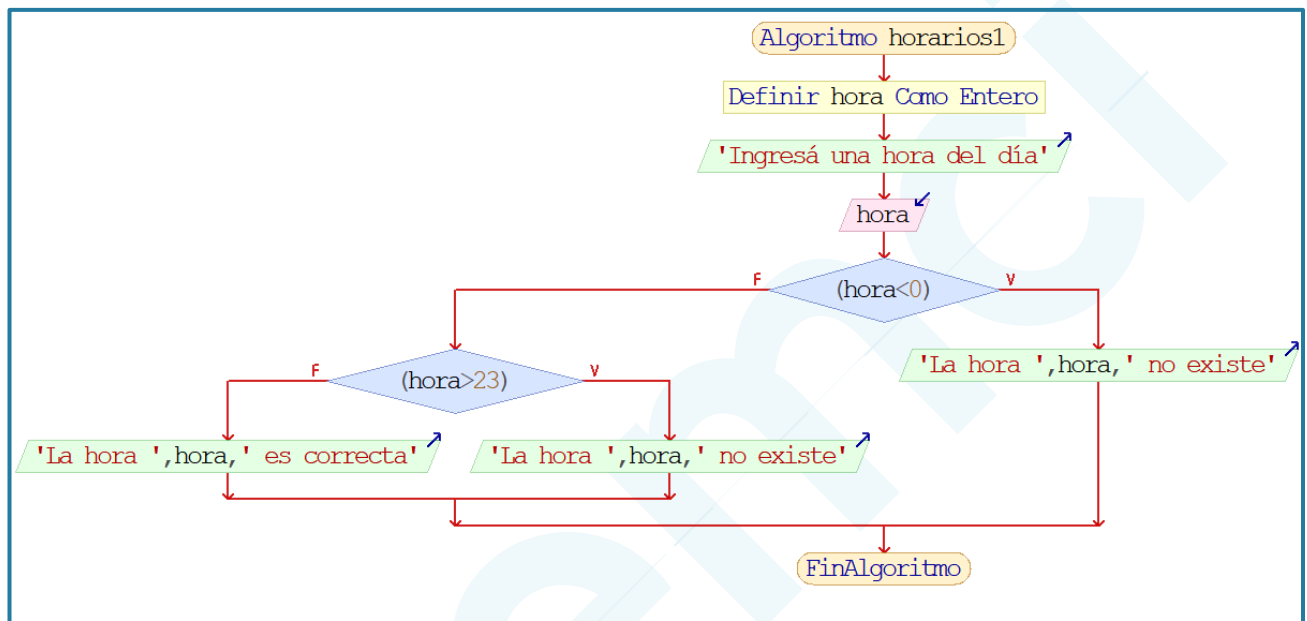


Ilustración 22: Primera parte del reconocedor del momento del día.

Una vez hecha esta validación, lo que sigue es saber a qué momento del día pertenece la hora ingresada, si es que resulta correcta. Esta vez la presentación será al revés, primero fíjate cómo quedaría el diagrama de flujo (aumentá el zoom para verlo bien):

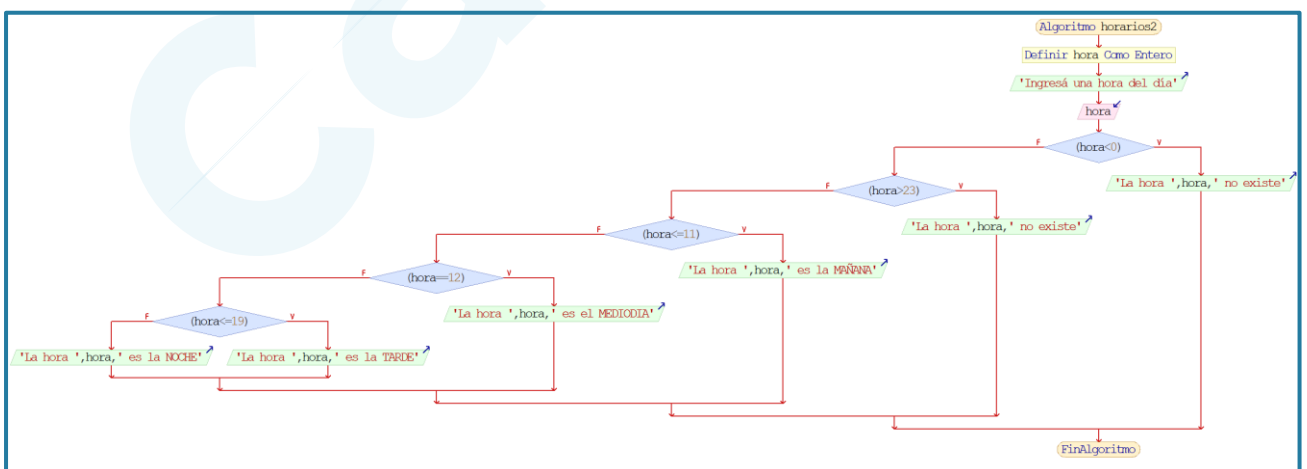


Ilustración 23: Segunda parte del reconocedor del momento del día.

Básicamente, reemplacé el mensaje **"La hora es correcta"** por un anidamiento de condiciones, que pasa por todas las posibilidades posibles buscando que alguna resulte **VERDADERO**. Si llegamos a la última condición y resulta **FALSO**, entonces por descarte podemos asegurar que se trata de la noche. Recordá que hay múltiples maneras de hacer esto. El orden de las condiciones es irrelevante siempre y cuando persiga la misma lógica.

El código queda de la siguiente manera:

```
1  Algoritmo horarios2
2      Definir hora Como Entero;
3      Escribir "Ingresá una hora del día";
4      Leer hora;
5      Si (hora<0) Entonces
6          Escribir "La hora ",hora," no existe";
7      SiNo
8          Si (hora>23) Entonces
9              Escribir "La hora ",hora," no existe";
10         SiNo
11             Si (hora<=11) Entonces
12                 Escribir "La hora ",hora," es la MAÑANA";
13             SiNo
14                 Si (hora==12) Entonces
15                     Escribir "La hora ",hora," es el MEDIODIA";
16                 SiNo
17                     Si (hora<=19) Entonces
18                         Escribir "La hora ",hora," es la TARDE";
19                     SiNo
20                         Escribir "La hora ",hora," es la NOCHE";
21                     FinSi
22                 FinSi
23             FinSi
24         FinSi
25     FinSi
26 FinAlgoritmo
```

Código 23: Segunda parte del algoritmo reconocedor del momento del día.

Probalo con cualquier valor entero y verás que funciona como corresponde. El problema que veo es que hay dos veces la misma instrucción: **Escribir "La hora ",hora," no existe";**

Como informático, si ves que algo se repite es porque no se está procediendo del todo bien. Hasta aquí, con las herramientas que te he mostrado, no hay alternativa más que repetir la instrucción, pues una hora del día puede ser inexistente según dos condiciones: que sea **menor a 0** o **mayor a 23**.

A continuación, verás una manera de optimizar esto y no repetir la instrucción.

Operadores lógicos

Son aquellos que permiten operar con expresiones booleanas. A través de ellos, pueden realizarse condiciones más complejas que no requieran un anidamiento de estructuras de selección.

Así como los **operadores aritméticos** tienen como operandos a **números**, los **operadores lógicos** tienen como operandos a **valores lógicos (VERDADERO o FALSO)**.

Así como los **operadores aritméticos** devuelven como resultado un **número**, los **operadores lógicos** devuelven como resultado un **valor lógico (VERDADERO o FALSO)**.

Operador	Nombre
!	NO lógico (NOT)
&	Y lógico (AND)
	O lógico (OR)

Cada uno de estos operadores tiene su tabla de verdad, es decir, una representación de todas las combinaciones posibles con sus correspondientes resultados. A continuación, te muestro en detalle cada operador.

Operador NOT

El operador **NOT**, también llamado **NO** o **negación lógica**, a diferencia de los que venías observando, es un operador **monádico**. Esto significa que trabaja con un solo operando. Lo que hace este operador es negar la expresión a la cual afecta. Se representa con el símbolo **!**.

Probá el siguiente código:

```

1  Algoritmo not
2      Escribir 2 < 3;
3      Escribir !(2 < 3);
4  FinAlgoritmo

```

Código 24: Demostración del operador NOT.

La expresión `2 < 3` es verdadera de aquí a la China, por lo tanto, el primer **Escribir** muestra **VERDADERO**. En cambio, la expresión lógica en el segundo **Escribir** está siendo negada, por lo tanto, se devuelve lo contrario (**FALSO**).

Los paréntesis en la segunda salida no son obligatorios, pero ayudan a la legibilidad.

La tabla de verdad del operador **NOT** es la siguiente:

Expresión booleana	Resultado
!VERDADERO	FALSO
!FALSO	VERDADERO

Operador OR

El operador **OR**, también llamado **O** o **disyunción lógica**, trabaja con dos operandos. Lo que hace este operador es devolver un valor **VERDADERO** si al menos un operando es **VERDADERO**, en caso contrario devuelve **FALSO**. Se representa con el símbolo `|`.

Es importante aclarar que, si bien PSeInt permite usar un solo `|`, en muchos lenguajes un operador `|` representa un **OR de bajo nivel**, es decir bit a bit, cuestión que no trataré, representando al operador **OR lógico** con un `||`. En este libro usaré el operador `|`.

Voy con un ejemplo. La condición para pasar a cierto evento es **ser mayor de edad o tener un nombre cuya longitud no supere los seis caracteres**. Haré un programa que permita al usuario ingresar su nombre y su edad. Luego la máquina informará si puede pasar o no, según la condición planteada.

El código puede ser el siguiente:

```

1  Algoritmo or
2      Definir nombre Como Cadena;
3      Definir edad Como Entero;

```



```

4      Escribir "Ingresá tu nombre";
5      Leer nombre;
6      Escribir "Ingresá tu edad";
7      Leer edad;
8      Si (Longitud(nombre) <= 6 | edad >= 18) Entonces
9          Escribir "Puede pasar";
10     SiNo
11         Escribir "NO Puede pasar";
12     FinSi
13 FinAlgoritmo

```

Código 25: Algoritmo que valida un pase según ciertas condiciones con el operador OR.

Mirá las siguientes pruebas:

Nombre	Edad	Salida
"Carlos"	25	"Puede pasar"
"María"	14	"Puede pasar"
"Ricardo"	40	"Puede pasar"
"Daniela"	8	"NO Puede pasar"

Para formalizar, la tabla de verdad del operador **OR** es la siguiente:

Expresión booleana	Resultado
VERDADERO VERDADERO	VERDADERO
VERDADERO FALSO	VERDADERO
FALSO VERDADERO	VERDADERO
FALSO FALSO	FALSO

Una buena aplicación práctica de este operador podría ser para pedirle una respuesta al usuario. Suponé una aplicación que requiera que el usuario ingrese la palabra **"Si"** para continuar. Es evidente que puede ingresarlo todo en minúsculas, todo en mayúsculas o de forma alternada. Para contemplar todas estas combinaciones y que la

máquina las interprete como la misma opción, podés armar una condición con operadores lógicos.

Mirá el siguiente código:

```
1  Algoritmo or2
2      Definir opcion Como Cadena;
3      Escribir "Ingrese su opción (SI/NO)";
4      Leer opcion;
5      Si (opcion == "si" | opcion == "SI" | opcion == "Si" | opcion == "sI") Entonces
6          Escribir "Pusiste que sí";
7      SiNo
8          Escribir "Pusiste otra respuesta";
9      FinSi
10 FinAlgoritmo
```

Código 26: Algoritmo que valida una opción.

Aunque personalmente yo hubiera elegido una manera menos engorrosa. Ya que te he enseñado a hacer uso de funciones, puedo pasar la opción a mayúsculas o a minúsculas y comparar solo una vez, quedando así:

```
1  Algoritmo or3
2      Definir opcion Como Cadena;
3      Escribir "Ingrese su opción (SI/NO)";
4      Leer opcion;
5      Si ( Minusculas(opcion) == "si" ) Entonces
6          Escribir "Pusiste que sí";
7      SiNo
8          Escribir "Pusiste otra respuesta";
9      FinSi
10 FinAlgoritmo
```

Código 27: Algoritmo alternativo que valida una opción.

Operador AND

El operador **AND**, también llamado **Y** o **conjunción lógica**, trabaja con dos operandos. Lo que hace este operador es devolver un valor **VERDADERO** si todos los operandos

resultan **VERDADERO**, en caso contrario devuelve **FALSO**. Se representa con el símbolo **&**.

Es importante aclarar que, si bien PSeInt permite usar un solo **&**, en muchos lenguajes un operador **&** representa un **AND de bajo nivel**, es decir bit a bit, cuestión que no trataré, representando al operador **AND lógico** con un **&**. En este libro usaré el operador **&**.

Seguiré con el mismo ejemplo anterior, pero esta vez cambiaré las reglas. Fijate como cambiar un **o** por un **y** hace una gran diferencia. La condición para pasar a cierto evento es **ser mayor de edad y tener un nombre cuya longitud no supere los seis caracteres**. Haré un programa que permita al usuario ingresar su nombre y su edad. Luego la máquina informará si puede pasar o no, según la condición planteada.

El código puede ser el siguiente:

```

1  Algoritmo and
2      Definir nombre Como Cadena;
3      Definir edad Como Entero;
4      Escribir "Ingresa tu nombre";
5      Leer nombre;
6      Escribir "Ingresa tu edad";
7      Leer edad;
8      Si ( Longitud(nombre) <= 6 & edad >= 18 ) Entonces
9          Escribir "Puede pasar";
10     SiNo
11         Escribir "NO Puede pasar";
12     FinSi
13 FinAlgoritmo
    
```

Código 28: Algoritmo que valida un pase según ciertas condiciones con el operador **AND**.

Mirá las siguientes pruebas:

Nombre	Edad	Salida
"Carlos"	25	"Puede pasar"
"María"	14	"NO Puede pasar"
"Ricardo"	40	"NO Puede pasar"

"Daniela"	8	"NO Puede pasar"
-----------	---	------------------

Para formalizar, la tabla de verdad del operador **AND** es la siguiente:

Expresión booleana	Resultado
VERDADERO & VERDADERO	VERDADERO
VERDADERO & FALSO	FALSO
FALSO & VERDADERO	FALSO
FALSO & FALSO	FALSO

Una buena aplicación práctica de este operador podría ser para pedirle al usuario un valor numérico perteneciente a cierto rango. Suponé que el usuario debe ingresar una nota numérica entre **0** y **10**. Podría tener tal regla en una sola condición.

Mirá el siguiente código:

1	Algoritmo and2
2	Definir num Como Entero;
3	Escribir "Ingrese un número entre 0 y 10 (inclusive)";
4	Leer num;
5	Si (num >= 0 & num <= 10) Entonces
6	Escribir "Número válido";
7	SiNo
8	Escribir "Número NO válido";
9	FinSi
10	FinAlgoritmo

Código 29: Algoritmo que valida que un número esté entre **0** y **10**.

Ahora podés dejar más elegante el algoritmo que calcula el momento del día según la hora. Al hacer uso de operadores lógicos, ahorrarás tener que escribir dos veces la misma instrucción acerca de que la hora no existe. Utilicé la condición:

(hora<0 | hora>23)

(¿la hora es menor a 0 o mayor a 23?) para saber si el número es inválido, pero también pude haber puesto:

!(hora>=0 & hora<=23)

(¿la hora no está entre 0 y 23?). Es equivalente.

```

1  Algoritmo horarios3
2      Definir hora Como Entero;
3      Escribir "Ingresá una hora del día";
4      Leer hora;
5      Si (hora<0 | hora>23) Entonces
6          Escribir "La hora ",hora," no existe";
7      SiNo
8          Si (hora<=11) Entonces
9              Escribir "La hora ",hora," es la MAÑANA";
10         SiNo
11             Si (hora==12) Entonces
12                 Escribir "La hora ",hora," es el MEDIODIA";
13             SiNo
14                 Si (hora<=19) Entonces
15                     Escribir "La hora ",hora," es la TARDE";
16                 SiNo
17                     Escribir "La hora ",hora," es la NOCHE";
18             FinSi
19         FinSi
20     FinSi
21 FinSi
22 FinAlgoritmo

```

Código 30: Algoritmo reconocedor del momento del día optimizado.

Tocá el botón  en PSeInt y observá como quedó el diagrama de flujo.

Jerarquía de operadores

Ya que han sido analizados todos los operadores con los que cuenta PSeInt, es importante que notes el orden en el que el intérprete evalúa y realiza las operaciones. Recordá que, para alterar este orden, podés hacer uso de paréntesis en cualquier expresión, en caso contrario, se evaluará de la siguiente manera:

Operadores	Nombres
()	Paréntesis.
$^ \%$	Potenciación y módulo.
$* /$	Multiplicación y división.
$+ -$	Suma y resta.
$< <= > >= == != <>$	Menor, menor o igual, mayor, mayor o igual, igual y distinto.
!	NOT lógico.
&	AND lógico.
	OR lógico.

Flujo de selección múltiple

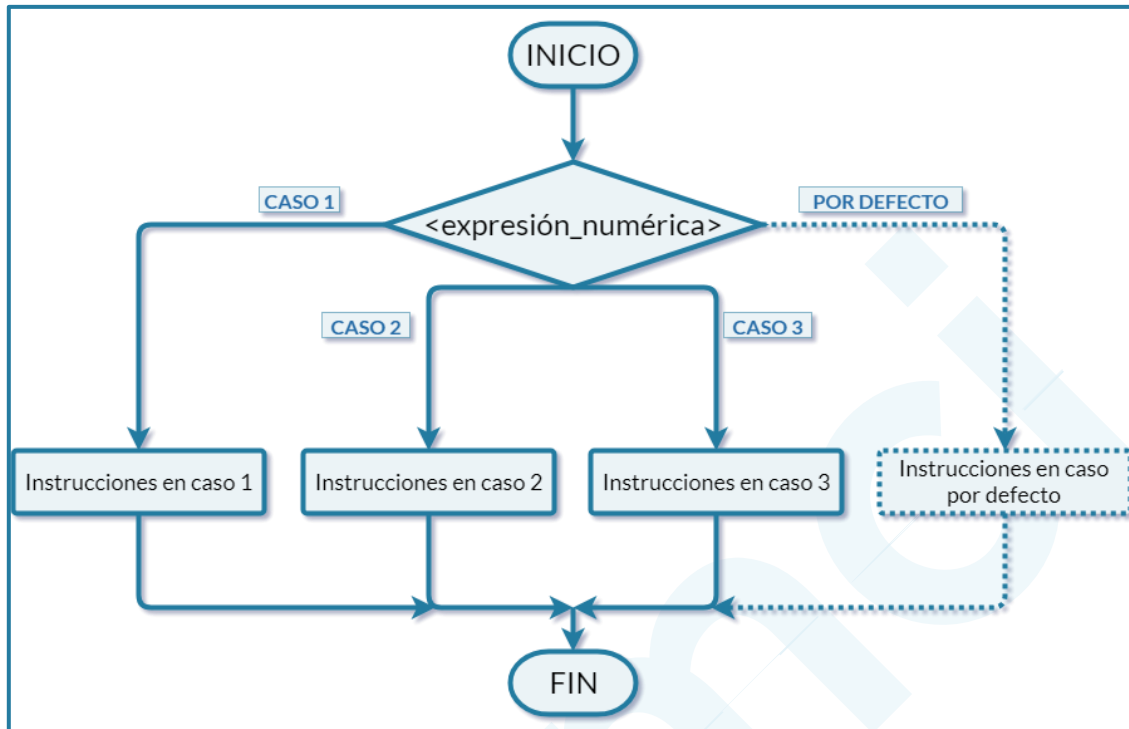


Ilustración 24: Flujo de selección múltiple.

Suponé que deseás realizar un menú de opciones, como cuando llamás por teléfono y la voz del otro lado te dice: *"Para ventas, presione 1. Para pagos, presione 2. Para servicio técnico, presione 3..."*. Realizar un menú con, por ejemplo, diez opciones diferentes, requiere, como habrás visto, realizar un anidamiento de estructuras de selección que puede tornarse inmanejable, engorroso y poco legible.

Para los casos en los que el dato a evaluar sea un número, y además por comparación (es decir, cada opción tiene un número concreto, no un rango de valores) se puede hacer uso de una estructura de selección múltiple, la cual presenta un diagrama como el siguiente:

La sintaxis es la siguiente:

Segun (<variable_numérica>) Hacer

<opcion>:

<instrucciones>

<opcion>:

<instrucciones>

<opcion>:

<instrucciones>

...

De Otro Modo:

<instrucciones>

FinSegun

- **Segun** se tipea literalmente, indica que vamos a iniciar un bloque de selección múltiple.
- **<variable_numérica>** es la variable que la computadora evaluará. El hecho de estar entre paréntesis es opcional, aunque te recomiendo que lo hagas para mejorar la legibilidad y porque muchos lenguajes formales requieren que la condición esté obligatoriamente encerrada entre paréntesis.
- **Hacer** se tipea literalmente, indica que a continuación serán listadas las posibilidades que puede tomar la **<variable_numérica>**.
- **<opcion>** es un valor numérico que se estima puede llegar a valer la **<variable_numérica>**. El **:** luego de cada **<opcion>** es obligatorio e indica que a continuación se listarán las instrucciones para ese caso.
- **<instrucciones>** son las instrucciones que se ejecutarán si la opción asociada se cumple.
- Los **...** indican que pueden seguir listándose posibles opciones, cada una con su correspondiente lista de instrucciones a ejecutarse.
- **De Otro Modo:** se tipea literalmente, aunque es opcional. Se listan las instrucciones que se ejecutarán en caso de que ninguna de las opciones contempladas ocurra.
- **FinSegun** se tipea literalmente, indica que ha terminado el bloque de selección múltiple.

Hagamos el menú, con las siguientes opciones:

1. Ventas.
2. Pagos.
3. Servicio técnico.
4. Gerencia.

Lo más conveniente es hacer uso de **Segun...FinSegun**:

1	Algoritmo menu_telefonico
2	Definir num Como Entero;

```
3      Escribir "MENU DE OPCIONES";
4      Escribir "[1] Ventas";
5      Escribir "[2] Pagos";
6      Escribir "[3] Servicio técnico";
7      Escribir "[4] Gerencia";
8      Escribir "Elegí tu opción:";
9      Leer num;
10     Segun (num) Hacer
11         1:
12             Escribir "Elegiste ventas";
13         2:
14             Escribir "Elegiste pagos";
15         3:
16             Escribir "Elegiste servicio técnico";
17         4:
18             Escribir "Elegiste gerencia";
19     FinSegun
20 FinAlgoritmo
```

Código 31: Algoritmo que simula un menú telefónico con una estructura de selección múltiple **Segun...FinSegun**.

Como verás, queda mucho más elegante que usar estructuras de selección doble anidadas.

El diagrama de flujo:

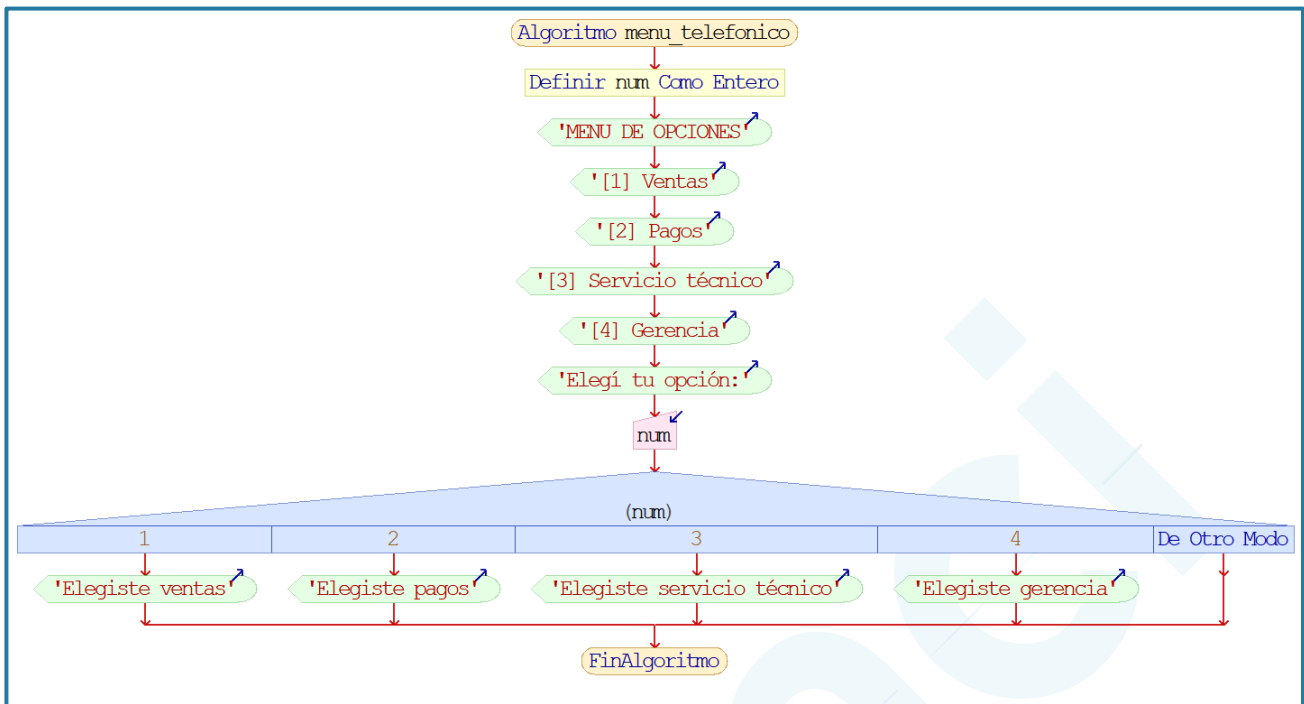


Ilustración 25: Algoritmo que simula un menú telefónico con una estructura de selección múltiple *Según...FinSegun*.

Si el usuario ingresa un valor no contemplado entre las opciones dentro del bloque *Segun...FinSegun*, el programa simplemente continúa su ejecución.

Poner una variable no numérica (como una cadena o un lógico) para ser evaluada por un bloque *Segun...FinSegun* derivará en un error en tiempo de ejecución.

En este caso, le agregaré a este ejemplo la posibilidad de que la máquina realice instrucciones en caso de que el número ingresado no sea ninguno de los listados. Para ello, haré uso de la sentencia *De Otro Modo*. La computadora responderá **"Incorrecto"** si se llega a tal caso.

```

1  Algoritmo menu_telefonico_mejorado
2      Definir num Como Entero;
3      Escribir "MENU DE OPCIONES";
4      Escribir "[1] Ventas";
5      Escribir "[2] Pagos";
6      Escribir "[3] Servicio técnico";
7      Escribir "[4] Gerencia";
8      Escribir "Elegí tu opción:";

```

```

9      Leer num;
10     Segun (num) Hacer
11         1:
12             Escribir "Elegiste ventas";
13         2:
14             Escribir "Elegiste pagos";
15         3:
16             Escribir "Elegiste servicio técnico";
17         4:
18             Escribir "Elegiste gerencia";
19     De Otro Modo:
20         Escribir "Incorrecto";
21     FinSegun
22     FinAlgoritmo

```

Código 32: Algoritmo que simula un menú telefónico contemplando opciones incorrectas con una estructura de selección múltiple **Según...FinSegun**.

El uso de la sentencia **De Otro Modo** es opcional.

Integrando los conceptos

A continuación, te presento un programa que integra todos los conceptos vistos hasta aquí. Se trata de un algoritmo que pide al usuario dos números y la computadora ofrece la posibilidad de operarlos matemáticamente según la elección del usuario:

```

1      Algoritmo calculadora
2          Definir a Como Real;
3          Definir b Como Real;
4          Definir opcion Como Entero;
5          Escribir "CALCULADORA";
6          Escribir "Ingrese número A";
7          Leer a;
8          Escribir "Ingrese número B";
9          Leer b;
10         Escribir "[1] A + B";

```

```

11  Escribir "[2] A - B";
12  Escribir "[3] A * B";
13  Escribir "[4] A / B";
14  Escribir "[5] Raíz cuadrada de A";
15  Escribir "[6] Raíz cuadrada de B";
16  Escribir "Elegí la operación:";
17  Leer opcion;
18  Si (opcion >= 1 & opcion <= 6) Entonces
19      Escribir "El resultado es " Sin Saltar;
20      Segun (opcion) Hacer
21          1:
22              Escribir a + b;
23          2:
24              Escribir a - b;
25          3:
26              Escribir a * b;
27          4:
28              Si (b == 0) Entonces
29                  Escribir "imposible de calcular.";
30              SiNo
31                  Escribir a / b;
32              FinSi
33          5:
34              Si (a >= 0) Entonces
35                  Escribir rc(a);
36              SiNo
37                  Escribir "imposible de calcular.";
38              FinSi
39          6:
40              Si (b >= 0) Entonces
41                  Escribir rc(b);
42              SiNo

```

```
43      Escribir "imposible de calcular.";
44      FinSi
45      FinSegun
46      SiNo
47          Escribir "Opción incorrecta. FIN";
48      FinSi
49      FinAlgoritmo
```

Código 33: Algoritmo que simula una calculadora básica.

Flujo de repetición

Tanto el flujo secuencial como el flujo de selección tienen en común que las instrucciones siempre siguen un curso hacia adelante, es decir, una vez que una instrucción fue ejecutada, jamás volverá a ejecutarse, a no ser, claro está, que el programador la repita más adelante.

Hay muchas situaciones donde se requiere repetir una o más instrucciones un número definido o no definido de veces, pero queriendo evitar que las instrucciones pertinentes sean repetidas en el código.

Las estructuras de repetición nos permiten solucionar esto. Se denominan **ciclos**, **bucles** o **loops**. Un ciclo puede tener una o más repeticiones, las cuales normalmente se denominan **vueltas** o **iteraciones**.

Hay tres componentes que deben ser correctamente analizados para poder efectuar ciclos:

- **Inicialización:** Cómo será el valor inicial de la(s) variable(s) en la condición.
- **Condición:** Qué tiene que ocurrir para que el ciclo continúe su ejecución. Debe tener al menos una variable involucrada, de lo contrario, el resultado sería constante.
- **Actualización:** La(s) variable(s) en la condición debe(n) cambiar su valor por cada iteración para que el resultado de la **condición** no sea siempre constante.

Flujo de repetición Mientras...FinMientras

Una estructura de repetición nos permite especificar que un programa debe repetir una o más acciones mientras cierta condición valga **VERDADERO**. Cuando la condición pase a valer **FALSO**, el programa continuará con la ejecución de las demás sentencias debajo de la estructura de control de repetición.

Supongamos que queremos escribirle al usuario la frase **"Hola, mucho gusto."** unas cinco veces. Es cierto que una posibilidad es usar cinco instrucciones **Escribir**, una debajo de otra con un resultado satisfactorio. Pero es evidente que esto no es muy práctico. Peor aún, imaginá que olvidaste poner un punto al final de la oración. ¿Te imaginás cambiándolo en las cinco instrucciones? ¿Y si hubiese querido mostrar diez mil oraciones? Aquí es cuando una estructura de repetición nos salva.

Observá el siguiente diagrama:

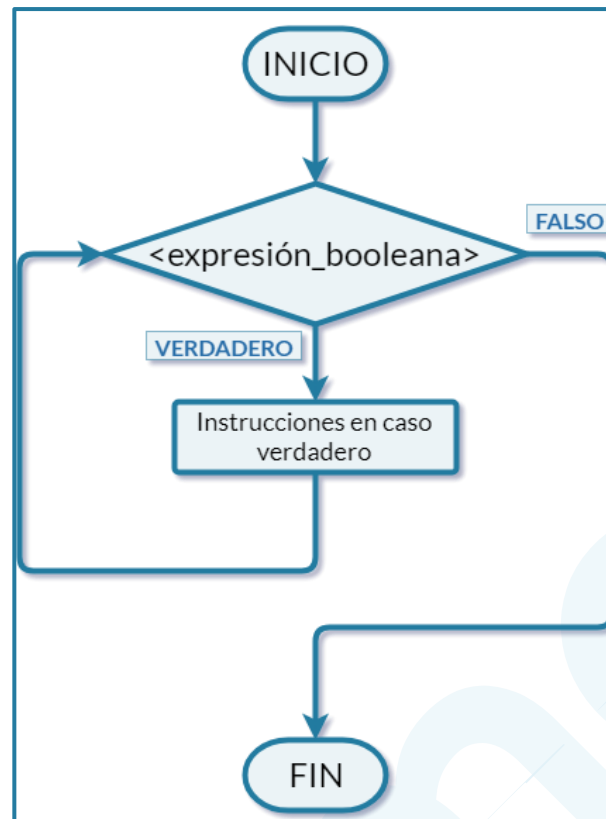


Ilustración 26: Flujo de repetición.

La sintaxis en pseudocódigo es la siguiente:

Mientras (<expresión_booleana>) **Hacer**
 <instrucciones>

FinMientras

- **Mientras** se tipea literalmente, indica que vamos a iniciar un bloque de repetición.
- **<expresión_booleana>** es la condición que la computadora evaluará. El hecho de estar entre paréntesis es opcional, aunque te recomiendo que lo hagas para mejorar la legibilidad y porque muchos lenguajes formales requieren que la condición esté obligatoriamente encerrada entre paréntesis.
- **Hacer** se tipea literalmente, indica que a continuación serán listadas las instrucciones que se ejecutarán en caso de que la condición sea **VERDADERO**.
- **<instrucciones>** son las instrucciones que se ejecutarán. Puede ser solo una, o varias, siguiendo los conceptos vistos hasta aquí.
- **FinMientras** se tipea literalmente, indica que ha terminado el bloque de repetición.

Volviendo al ejemplo de mostrarle al usuario la frase **"Debo aprender ciclos."** cinco veces, el código puede ser el siguiente:

```

1  Algoritmo salidas_repetidas
2      Definir veces Como Entero;
3      veces = 0; // La inicialización
4      Mientras (veces < 5) Hacer // La condición
5          Escribir "Debo aprender ciclos.";
6          veces = veces + 1; // La actualización
7      FinMientras
8      Escribir "Fin.";
9  FinAlgoritmo
    
```

Código 34: Algoritmo que muestra cinco veces una cadena.

Resultando en el siguiente diagrama de flujo:

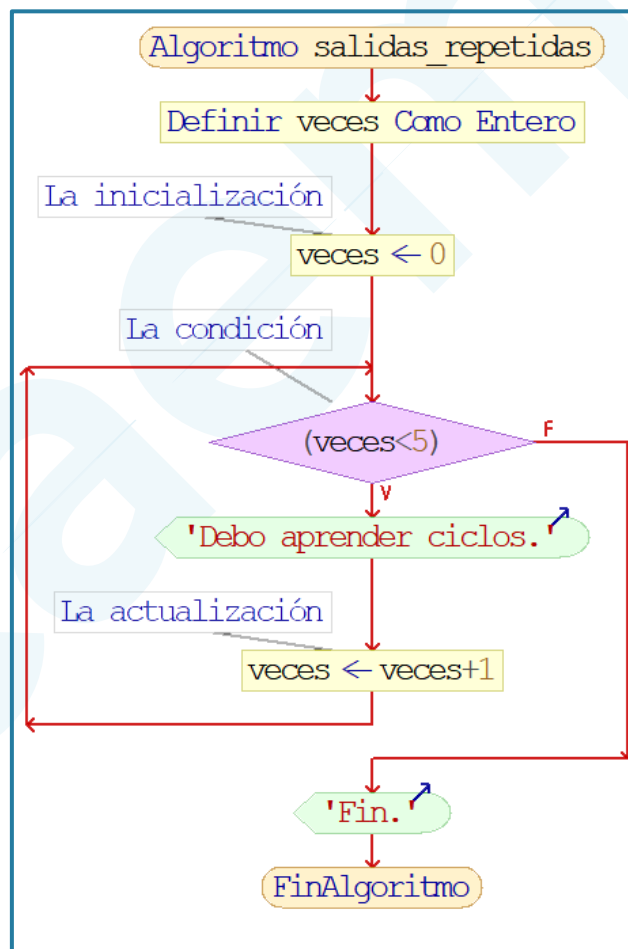


Ilustración 27: Algoritmo que muestra cinco veces una cadena.

Lo primero que hice fue definir una variable entera llamada **veces**, cuyo valor inicial es **0**. Es muy importante que **veces** tenga un valor inicial, pues a continuación, su valor es comparado con el número **5**. Si el valor de **veces** es menor que **5**, se ejecutan las instrucciones dentro del bloque **Mientras...FinMientras**. Dentro del bloque, se escribe la cadena **"Hola, mucho gusto."**. Acto seguido, se incrementa una unidad el valor de **veces**, a través de su valor actual. Si tenés problemas para entender por qué **veces** se repite a ambos lados del operador de asignación, te recomiendo que repases **Actualizar una variable** en la página 38.

Una vez que se llega a la sentencia **FinMientras**, el flujo vuelve hacia arriba a evaluar la condición. El valor de **veces** ya no es **0**, porque fue actualizado antes de terminar la primera iteración. Ahora su valor es **1**. Como el valor **1** es menor que **5**, se vuelven a realizar las instrucciones: mostrar la cadena e incrementar el valor de **veces**. Esto se repite hasta el caso donde **veces** valga **5**. Cuando la computadora evalúe la condición (**veces < 5**), el resultado será **FALSO**. En tal situación, el bloque de repetición termina, continuando con el normal flujo secuencial.

La ausencia o el mal cálculo de alguno de los tres componentes que protagonizan un ciclo (inicialización, condición y actualización) pueden provocar resultados inesperados: quizás se realicen más o menos iteraciones de las que fueron previstas, que el ciclo nunca se ejecute (la primera iteración resulta **FALSO**) o que el ciclo se ejecute infinitamente (todas las iteraciones resultan **VERDADERO**).

Es importante que notes que:

- En la **inicialización**, podés probar estableciendo el valor de **veces** en **1**, para que observes que solo se imprime cuatro veces la cadena **"Hola, mucho gusto."**.
- En la **condición**, podés probar cambiando el operador **<** por un **>**, para que observes que el ciclo nunca se ejecutará, pues la primera vez, **veces** vale **0** y tal valor no es mayor que **5**, haciendo que la **condición** resulte **FALSO** y continuando por debajo de la estructura de repetición.
- En la **actualización** podés probar borrar o poner en comentario la instrucción **veces = veces + 1**; para que observes que el ciclo se ejecuta infinitamente, debido a que **veces** vale constantemente **0**, haciendo que la **condición** resulte **VERDADERO** en todas las iteraciones.

Podrás ver que el uso de estructuras de repetición requiere de una atención especial. Es muy fácil que nuestro programa no tenga errores de sintaxis, pero sí lógicos, es decir, que haya cosas que no funcionen como esperamos o que den resultados inesperados. Como todo, es cuestión de tomarle la mano. Gracias a los ciclos, mostrar una frase cinco, diez, cien o un millón de veces es tan simple como cambiar el número con el cual se compara **veces** en la **condición**.

Control de ciclos por contador

En los casos donde se necesita la ejecución de un número definido de iteraciones, se recurre a un **ciclo controlado por contador**.

Una variable entera oficia de contadora de repeticiones. Comenzará con un valor inicial. Cada iteración actualizará el valor de la variable contadora. El ciclo seguirá iterando mientras la variable contadora supere o no (según el caso) a cierto valor.

```
1  Algoritmo salidas_repetidas_numeradas
2      Definir veces Como Entero;
3      veces = 0; // La inicialización
4      Mientras (veces < 5) Hacer // La condición
5          Escribir "Iteración " , veces , ": "Debo aprender ciclos.";
6          veces = veces + 1; // La actualización
7      FinMientras
8      Escribir "Fin.";
9  FinAlgoritmo
```

Código 35: Algoritmo que muestra cinco veces una cadena contando hacia adelante.

En el ejemplo anterior ya se ha usado una variable contadora, la cual llamamos **veces**. Para ver explícitamente cómo **veces** se trata de un **contador**, podemos hacer que en cada iteración se muestre el valor de **veces**, junto a la cadena **"Hola, mucho gusto."**. De esta manera, lograrás ver los resultados en pantalla numerados:

Como verás, ahora se muestra la frase precedida por el número de iteración correspondiente. Como **veces** se inicializó en **0**, es normal ver que las iteraciones mostradas vayan de **0** a **4**. Lo importante es que el objetivo se ha cumplido: mostrar la cadena **5** veces.

Hay, desde ya, infinitas maneras de realizar el mismo ejercicio. ¿Qué tal si hacemos la recorrida en sentido contrario? Esta vez, la variable **veces** se inicializa en **5**. El ciclo va a ejecutarse mientras **veces** sea mayor que **0**. La actualización esta vez consiste en decrementar **veces** en una unidad cada vez que se itera.

El código quedaría así:

```
1  Algoritmo salidas_repetidas_numeradas_reversa
2      Definir veces Como Entero;
3      veces = 5; // La inicialización
4      Mientras (veces >= 1) Hacer // La condición
```

```

5      Escribir "Iteración " , veces , ": Debo aprender ciclos.";
6      veces = veces - 1; // La actualización
7      FinMientras
8      Escribir "Fin.";
9  FinAlgoritmo

```

Código 36: Algoritmo que muestra cinco veces una cadena contando hacia atrás.

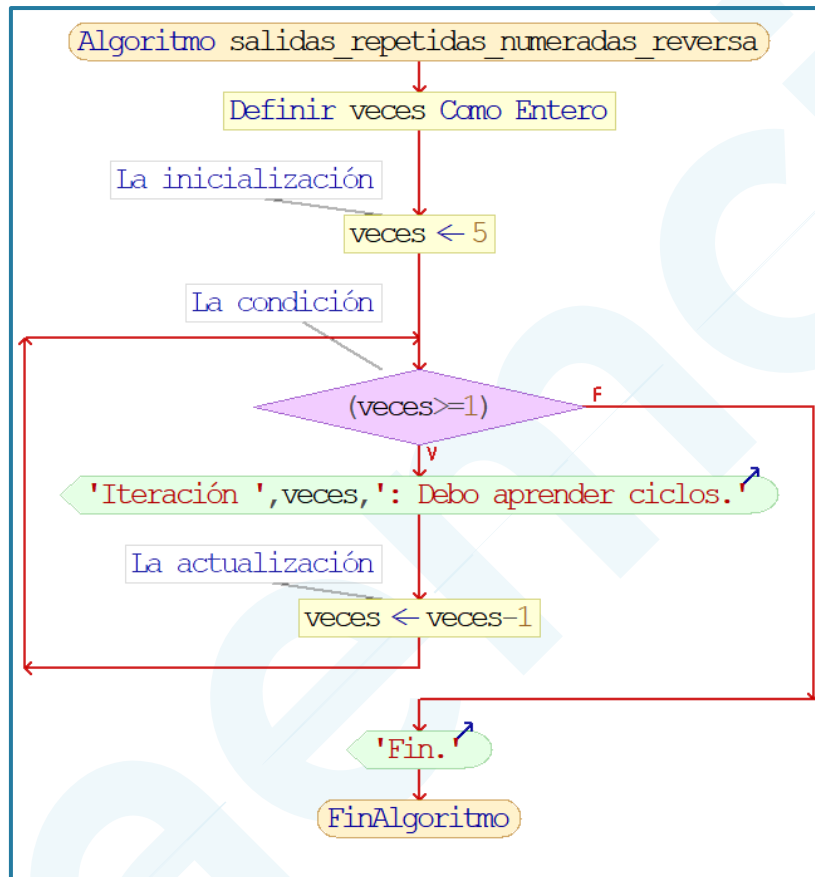


Ilustración 28: Algoritmo que muestra cinco veces una cadena contando hacia atrás.

El resultado es el mismo, se muestra 5 veces la cadena, pero observá que las iteraciones fueron contadas en orden decreciente.

Un último ejemplo. Mostrar los primeros veinte números pares, en líneas diferentes. Si un número es a la vez múltiplo de 6, informarlo. Este ejercicio requiere el uso de una estructura de selección dentro del ciclo.

```

1  Algoritmo muestra_veinte_pares
2      Definir num Como Entero;
3      num = 0; // La inicialización

```

```

4      Mientras (num <= 20) Hacer // La condición
5          Si (num % 6 == 0) Entonces // Si es múltiplo de 6
6              Escribir num , " (múltiplo de 6)";
7          SiNo
8              Escribir num;
9          FinSi
10         num = num + 2; // La actualización
11     FinMientras
12 FinAlgoritmo

```

Código 37: Algoritmo que muestra los primeros 20 números pares, aclarando cuáles son múltiplos de 6.

El diagrama de flujo quedaría así:

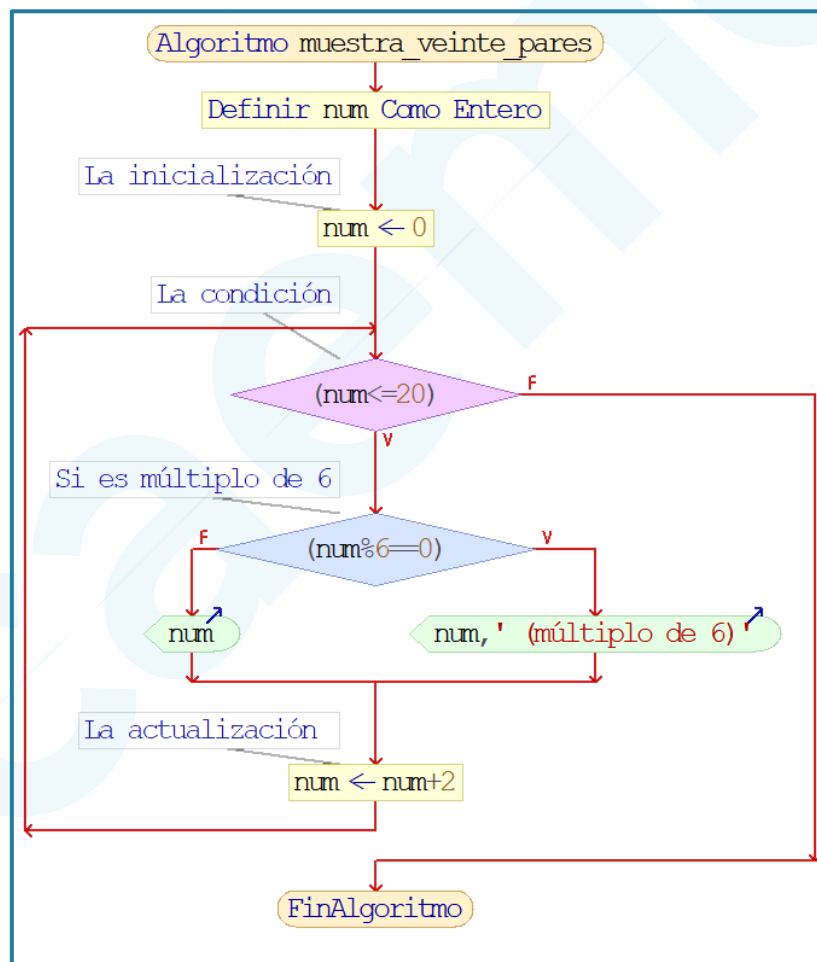


Ilustración 29: Algoritmo que muestra los primeros 20 números pares, aclarando cuáles son múltiplos de 6.

Acumuladores

Un **acumulador** es una variable que se utiliza para ir almacenando resultados parciales a medida que se llevan a cabo iteraciones. Es necesario que, por cada una de las iteraciones, se actualice el acumulador, para poder entregar los resultados correctos una vez que finalice el bucle.

Un ejemplo podría ser que la computadora muestre la suma de los primeros **10** números naturales, es decir, el resultado de **1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10**.

Si bien se puede escribir esa expresión como está, haciendo que la máquina directamente entregue el resultado, la idea es utilizar un ciclo que itere **10** veces. Por cada iteración, aprovecharemos el valor del **contador** para realizar la suma con lo que hasta ese momento es el resultado parcial, guardado en un **acumulador**.

El diagrama de flujo quedaría así:

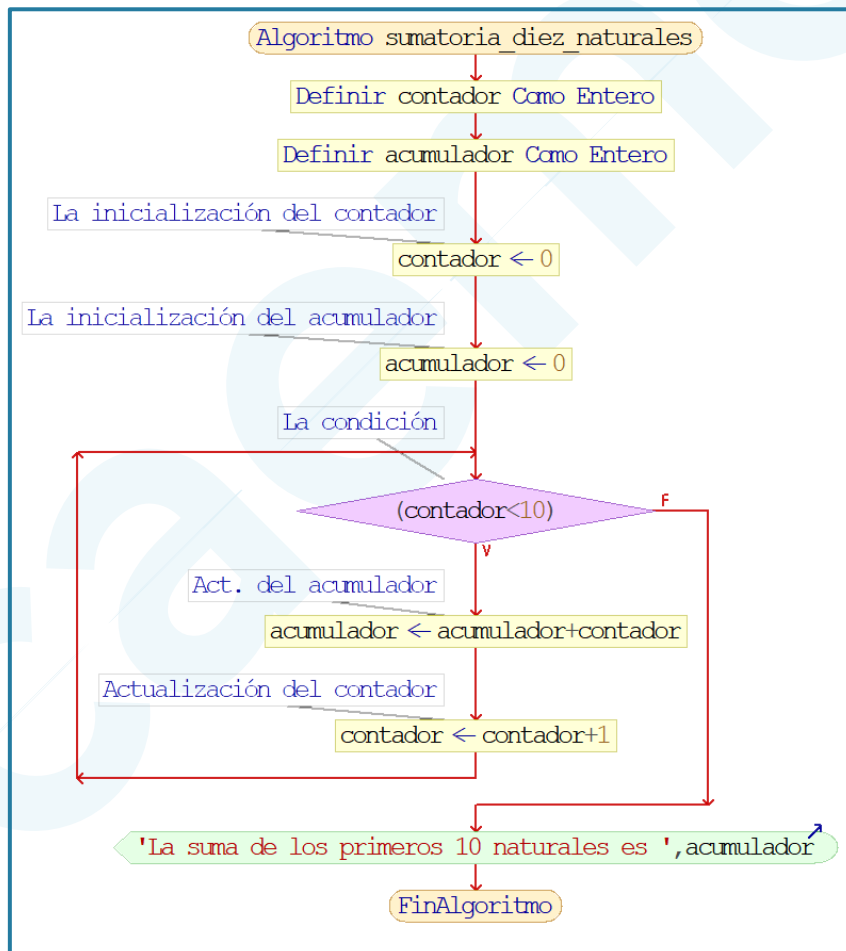


Ilustración 30: Algoritmo que calcula la sumatoria de los primeros **10 números naturales.**

El código sería el siguiente:

```

1  Algoritmo sumatoria_diez_naturales
2      Definir contador Como Entero;
3      Definir acumulador Como Entero;
4      contador = 0; // La inicialización del contador
5      acumulador = 0; // La inicialización del acumulador
6      Mientras (contador < 10) Hacer // La condición
7          acumulador = acumulador + contador; // Act. del acumulador
8          contador = contador + 1; // Actualización del contador
9      FinMientras
10     Escribir "La suma de los primeros 10 naturales es " , acumulador;
11 FinAlgoritmo

```

Código 38: Algoritmo que calcula la sumatoria de los primeros 10 números naturales.

Un programa algo más dinámico podría permitirle al usuario ingresar valores y que la computadora calcule la sumatoria de los mismos. En este caso tendríamos una instrucción de entrada dentro del ciclo. Además, podríamos hacer que sea el propio usuario quien indique la cantidad de números que va a ingresar.

El código sería el siguiente:

```

1  Algoritmo sumatoria_numeros_usuario
2      Definir contador Como Entero;
3      Definir acumulador Como Entero;
4      Definir cantNumeros Como Entero;
5      Definir num Como Entero;
6      contador = 0; // La inicialización del contador
7      acumulador = 0; // La inicialización del acumulador
8      Escribir "¿Cuántos números querés ingresar?";
9      Leer cantNumeros;
10     Mientras (contador < cantNumeros) Hacer // La condición
11         Escribir "Ingresá el " , (contador+1) , "º número";
12         Leer num;
13         acumulador = acumulador + num; // Act. del acumulador
14         contador = contador + 1; // Actualización del contador
15     FinMientras

```


16	Escribir "La suma es " , acumulador;
17	FinAlgoritmo

Código 39: Algoritmo que calcula la sumatoria de los números ingresados por el usuario.

El hecho de poner en la línea 11 la expresión **(contador+1)** no hará que **contador** modifique su valor. Es simplemente para que el usuario vea el orden de los números ingresados desde 1, y no desde 0, como internamente se estableció a **contador**. La verdadera **actualización** se produce en la instrucción **contador = contador + 1;**.

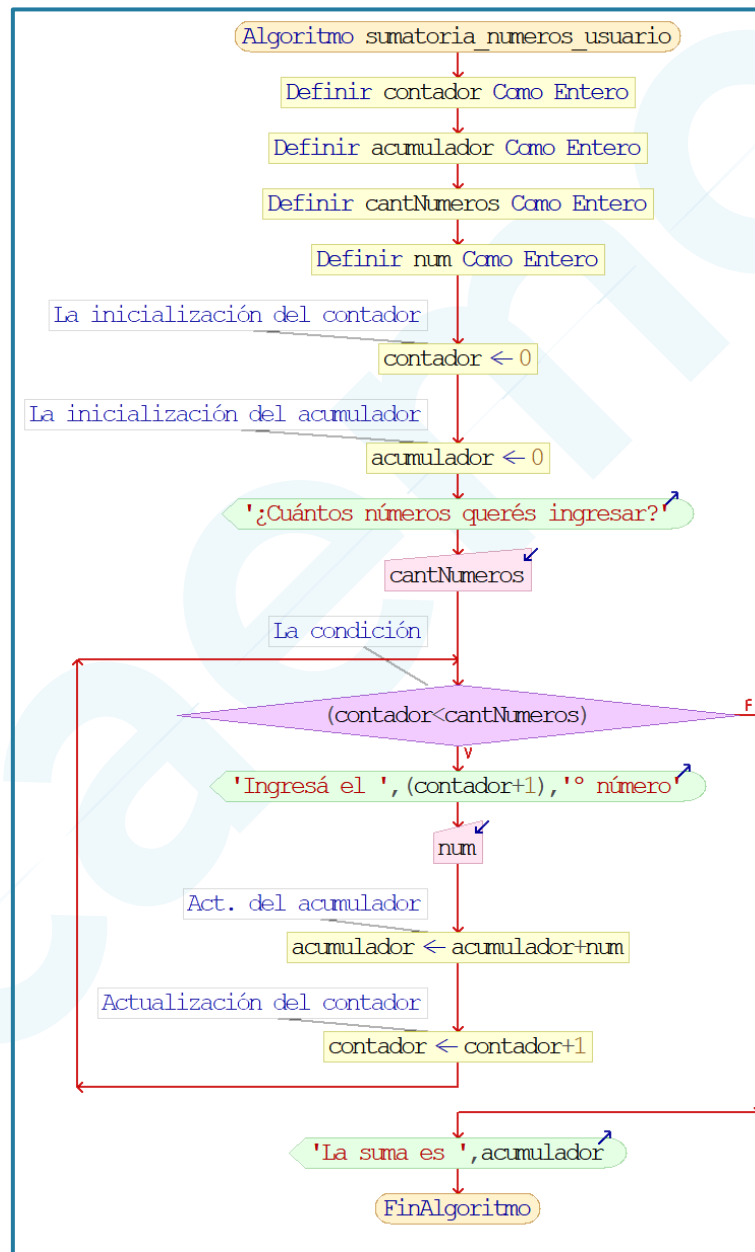


Ilustración 31: Algoritmo que calcula la sumatoria de los números ingresados por el usuario.

Pruebas de escritorio

Ahora que las instrucciones pueden llegar a repetirse, es probable que te desorientes un poco acerca del valor de las variables y lo que el programa realiza.

Una manera de comprobar que el programa funciona según lo estimado y, en caso de que no lo haga, identificar y corregir errores es realizar una **prueba de escritorio**.

Una **prueba de escritorio** consiste en hacer un seguimiento paso a paso del programa, tal como lo hace la máquina. En cada uno de estos pasos, se debe analizar el valor que toma cada variable y anotarlo (generalmente en una tabla), para poder llevar un control de lo que hace el programa. De esta manera, podemos detectar en qué punto específico se produce un error lógico (es decir, se toma un valor que no era el esperado) que arrastre a las instrucciones siguientes, provocando resultados inesperados.

La idea es anotar los nombres de las variables actuantes en el programa en distintas columnas. Por cada instrucción, se establece una nueva fila, anotando los valores de las variables tras esa ejecución.

Voy a efectuar el análisis del algoritmo que calcula la sumatoria de los primeros diez números naturales. El código es el siguiente:

```

1  Algoritmo sumatoria_diez_naturales
2      Definir contador Como Entero;
3      Definir acumulador Como Entero;
4      contador = 0; // La inicialización del contador
5      acumulador = 0; // La inicialización del acumulador
6      Mientras (contador < 10) Hacer // La condición
7          acumulador = acumulador + contador; // Act. del acumulador
8          contador = contador + 1; // Actualización del contador
9      FinMientras
10     Escribir "La suma de los primeros 10 naturales es " , acumulador;
11 FinAlgoritmo
  
```

Código 40: Algoritmo que calcula la sumatoria de los primeros 10 números naturales.

Voy a utilizar una tercera columna que muestre el valor que toma la condición, para que se note cuándo termina el ciclo de iterar.

contador	acumulador	contador < 10
0	0	VERDADERO

1	0	VERDADERO
2	1	VERDADERO
3	3	VERDADERO
4	6	VERDADERO
5	10	VERDADERO
6	15	VERDADERO
7	21	VERDADERO
8	28	VERDADERO
9	36	VERDADERO
10	45	FALSO

Como ves, cuando el contador vale **10**, la condición `contador < 10` resulta con valor **FALSO**. El último valor de acumulador fue **45**, siendo ese el número que se muestra en la salida.

Te invito a que realices la prueba de escritorio de todos los algoritmos realizados en este libro, con énfasis en los que tienen estructuras de repetición.

Control de ciclos por bandera

En los **ciclos controlados por contador**, el número de iteraciones está definido, ya sea por elección del programador o por el propio usuario. A continuación, verás una estrategia que define ciclos con un número de iteraciones no definidas de antemano.

En los **ciclos controlados por bandera**, el número de iteraciones no está definido, sino que depende de un valor normalmente llamado **valor de bandera** o **valor centinela**.

En el algoritmo que calcula la sumatoria de números ingresados por el usuario donde se utiliza un ciclo controlado por contador, el usuario debe primero ingresar un valor que representa la cantidad de números que posteriormente vaya a ingresar. El ciclo se ejecuta mientras el contador de iteraciones sea menor que este valor.

La idea es rehacer este programa utilizando un **ciclo controlado por bandera**.

El usuario no va a especificar el número de valores que vaya a ingresar, sino que directamente comenzará a introducirlos hasta que se detecte cierto valor que

consideraré como **valor de corte** o **valor de bandera**, que, en este caso, será el número **0**. Cuando se detecte el **valor de corte**, el ciclo finalizará.

En este tipo de estrategia, no está definido el número de iteraciones, sino que el usuario va decidiendo continuar o no, según los valores que ingrese.

El código es el siguiente:

```
1  Algoritmo sumatoria_numeros_usuario_bandera
2      Definir acumulador Como Entero;
3      Definir bandera Como Entero;
4      Definir num Como Entero;
5      acumulador = 0; // La inicialización del acumulador
6      bandera = 0; // Valor de corte
7      num = 1; // Puede ser cualquiera distinto del corte
8      Mientras (num != bandera) Hacer // La condición
9          Escribir "Ingresá un número (Para terminar, ingresá 0)";
10         Leer num;
11         acumulador = acumulador + num; // Act. del acumulador
12     FinMientras
13     Escribir "La suma es " , acumulador;
14 FinAlgoritmo
```

Código 41: Algoritmo que calcula la sumatoria de números ingresados por el usuario utilizando ciclo controlado por bandera.

La elección del valor de corte no es al azar. He considerado al **0** porque es muy raro que alguien quiera agregar un **0** a una sumatoria, sabiendo que no modifica el resultado en absoluto. Eso mismo además hace que no deba preocuparme por el hecho de que **num** tome el valor **0** y se acumule en lo que sería la última iteración. El valor final es correcto.

Como en la estructura de repetición **Mientras...FinMientras** la condición se evalúa al principio, es obligatorio inicializar la variable que aloja el número ingresado por el usuario con un valor distinto del de corte. Este valor luego será reemplazado por el que ingrese el usuario en la primera iteración.

Quiero que notes que **num** pudo haber sido inicializado con cualquier valor (yo elegí un **1**), excepto el **0**, ya que en tal caso, la condición **num != bandera** hubiera dado **FALSO**, haciendo que el ciclo jamás se ejecute.

El diagrama de flujo queda de la siguiente forma:

Si el programa se tratase de ingresar calificaciones numéricas, elegiría como valor de corte el **-1**, que es evidente que no corresponde a ningún valor válido de calificación. Pero debo tener cuidado cuando el ciclo termina, pues se acumuló un **-1** en la última iteración que me alteraría al **acumulador**. Antes de realizar cualquier cálculo o salida, debería sumar un **1** al **acumulador**.

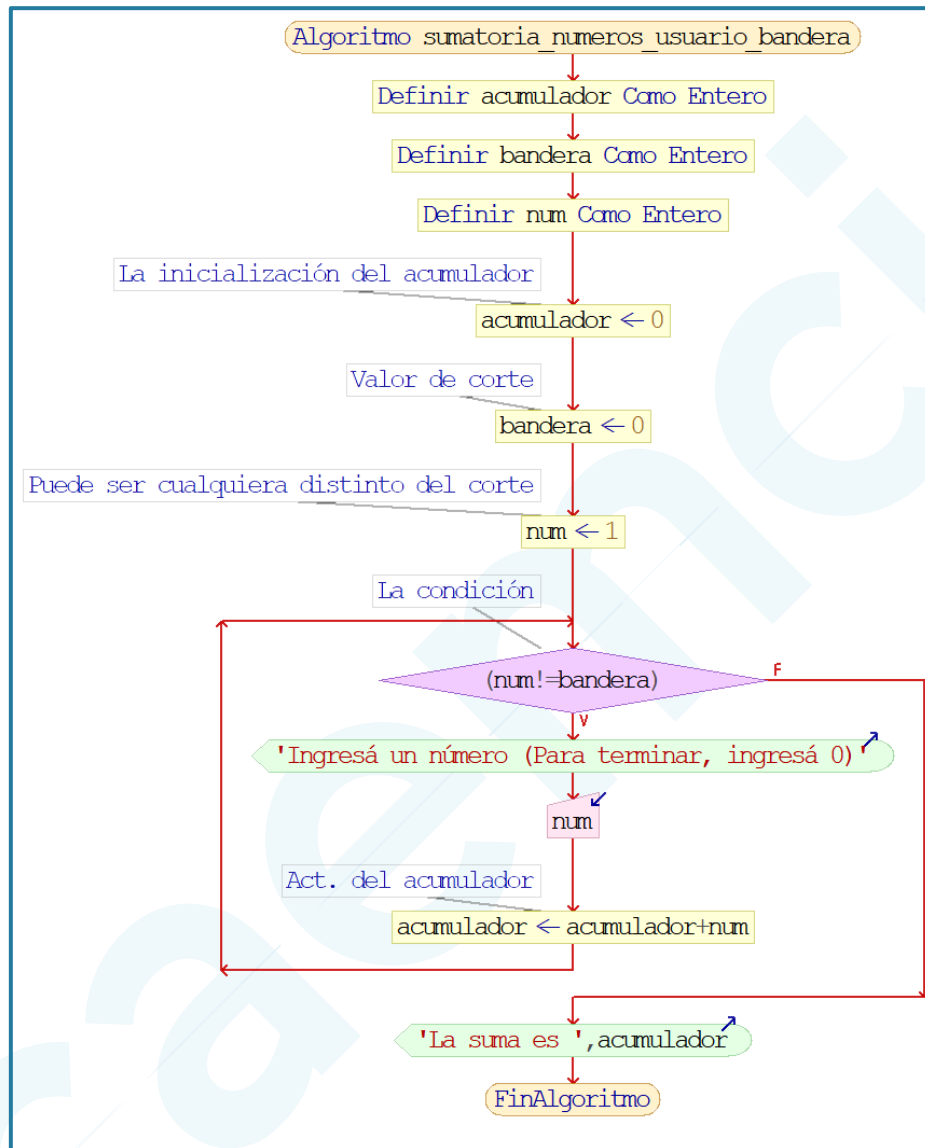


Ilustración 32: Algoritmo que calcula la sumatoria de números ingresados por el usuario utilizando ciclo controlado por bandera.

Otra manera de realizar este ejercicio es preguntarle al usuario si desea seguir ingresando datos cada vez que introduce un valor. Si el usuario decide continuar, el ciclo continúa iterando, en caso contrario, el ciclo termina. La manera de saber si el usuario eligió seguir, es que este presione la tecla **"s"** (de sí). Esta forma nos asegura que no habrá valores de corte que alteren mis resultados, pero para el usuario el ingreso de datos resulta más engorroso.

El código quedaría así:

```
1  Algoritmo sumatoria_numeros_usuario_bandera2
2      Definir acumulador Como Entero;
3      Definir num Como Entero;
4      Definir opc Como Cadena;
5      acumulador = 0; // La inicialización del acumulador
6      opc = "s"; // Debe ser la opción afirmativa para comenzar
7      Mientras (opc == "s" | opc == "S") Hacer // La condición
8          Escribir "Ingresá un número";
9          Leer num;
10         acumulador = acumulador + num; // Act. del acumulador
11         Escribir "¿Deseás seguir ingresando datos? [S/N]";
12         Leer opc;
13     FinMientras
14     Escribir "La suma es " , acumulador;
15 FinAlgoritmo
```

Código 42: Algoritmo alternativo que calcula la sumatoria de números ingresados por el usuario utilizando ciclo controlado por bandera.

El programa funciona como estaba previsto, pero quiero que notes que si el usuario ingresa cualquier valor distinto de "s" y "S", el ciclo terminará. La idea es que el ciclo termine cuando el valor ingresado sea "n" o "N". ¿Se puede solucionar esto?

A continuación, verás cómo.

Anidamiento de estructuras de repetición

Voy a tomar el ejemplo anterior con una reformulación para que el ciclo continúe cuando el valor ingresado sea "s" o "S" y el ciclo termine cuando el valor ingresado sea "n" o "N". En cualquier otro caso, se debe desestimar la opción y volver a preguntar.

En este caso se necesitan dos ciclos, uno dentro de otro. Se mantiene el ciclo que pide números hasta detectar "n" o "N" pero, además, dentro aparece un ciclo que pregunta si se desean ingresar más valores. La condición para salir de este último ciclo es que lo ingresado por el usuario sea una opción afirmativa o negativa. Cualquier valor no esperado, hará que el programa repregunte infinitamente hasta que se introduzca una opción válida.

Primero voy a compartirte el diagrama de flujo:

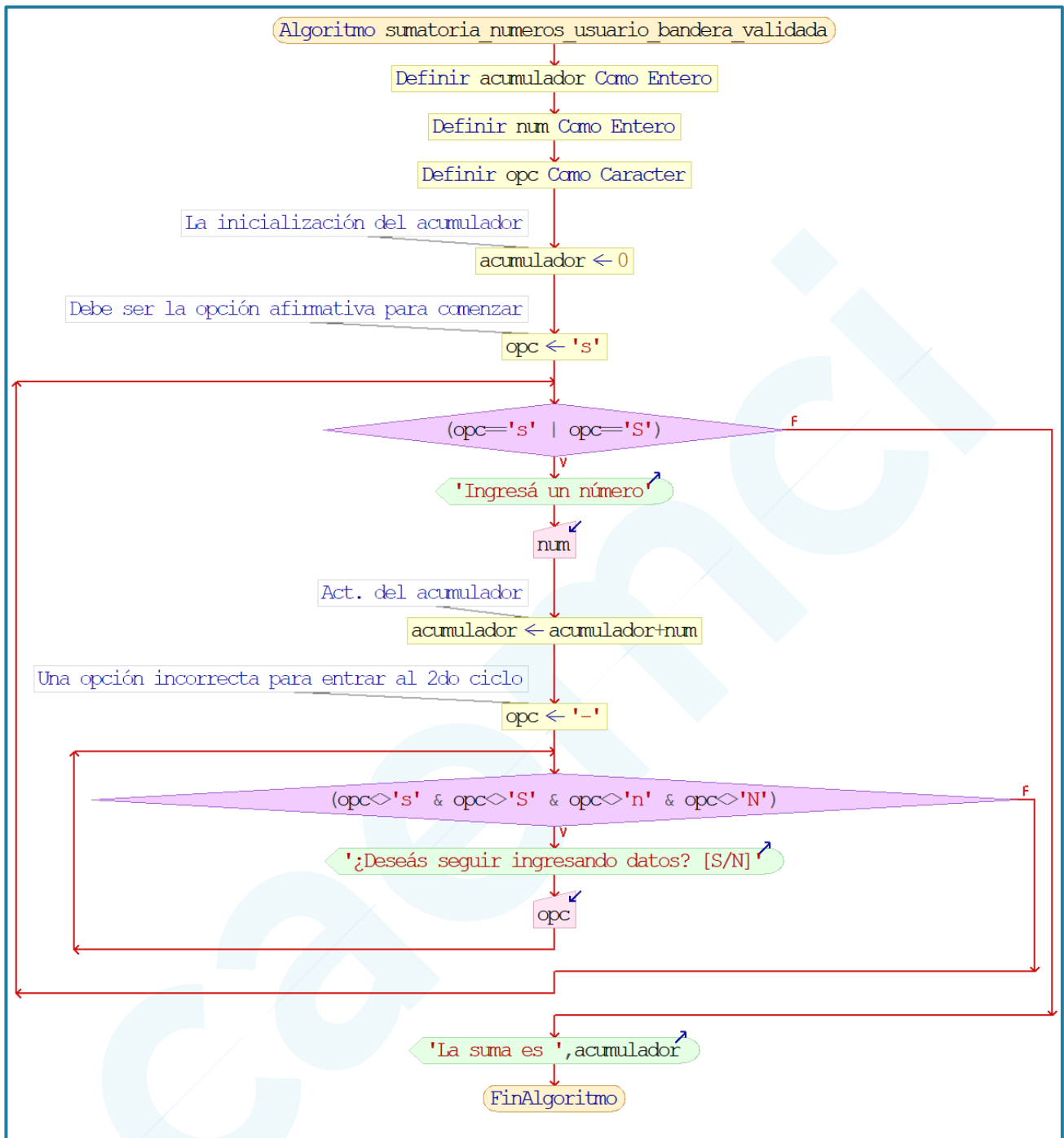


Ilustración 33: Algoritmo que calcula la sumatoria de números con validación de opción.

El código sería el siguiente:

1	Algoritmo sumatoria_numeros_usuario_bandera_validada
2	Definir acumulador Como Entero;
3	Definir num Como Entero;

```

4      Definir opc Como Cadena;
5      acumulador = 0; // La inicialización del acumulador
6      opc = "s"; // Debe ser la opción afirmativa para comenzar
7      Mientras (opc == "s" | opc == "S") Hacer
8          Escribir "Ingresá un número";
9          Leer num;
10         acumulador = acumulador + num; // Act. del acumulador
11         opc = "-"; // Una opción incorrecta para entrar al 2do ciclo
12         Mientras (opc <> "s" & opc <> "S" & opc <> "n" & opc <> "N") Hacer
13             Escribir "¿Deseás seguir ingresando datos? [S/N]";
14             Leer opc;
15         FinMientras
16     FinMientras
17     Escribir "La suma es " , acumulador;
18 FinAlgoritmo

```

Código 43: Algoritmo que calcula la sumatoria de números ingresados por el usuario con validación de opciones.

Voy a realizar un ejemplo más. Un algoritmo que pida al usuario dos dimensiones enteras positivas, **alto** y **ancho**. La computadora dibuja un rectángulo con asteriscos en la consola de las dimensiones que el usuario determinó.

¿Cómo hacer, ante todo, para validar que el usuario haya ingresado un valor entero y no real? Recordá que, si se asigna un valor real en una variable declarada entera, el programa generará un error en tiempo de ejecución.

La estrategia es alojar el número ingresado por el usuario en una variable declarada como real. Si el usuario ingresa un entero, no habría problemas: todo entero es un real. A partir de allí se pueden truncar los decimales con la función **trunc()** y guardar el valor en una variable entera. Eso haría que, si el usuario ingresa, por ejemplo, el valor **4.3**, el algoritmo lo procese y reconozca como un **4**. La otra alternativa es volver a pedirle la dimensión al usuario hasta que la misma sea válida, utilizando ciclos. Me decantaré por la última opción, ya que el motivo de este apartado es mostrar estructuras de repetición anidadas. Además, validaré que la dimensión ingresada sea mayor que **0**.

El código sería el siguiente:

```

1      Algoritmo dibujador_rectangulo
2          Definir ancho Como Real;

```



```

3      Definir alto Como Real;
4      Definir filas Como Entero; // Contador de filas
5      Definir columnas Como Entero; // Contador de columnas
6      filas = 0;
7      columnas = 0;
8      ancho = 0.1; // Un valor que haga entrar al ciclo
9      Mientras (trunc(ancho) != ancho | ancho <= 0) Hacer
10         Escribir "Ingrese ancho (valor entero)";
11         Leer ancho;
12      FinMientras
13      alto = 0.1; // Un valor que haga entrar al ciclo
14      Mientras (trunc(alto) != alto | alto <= 0) Hacer
15         Escribir "Ingrese alto (valor entero)";
16         Leer alto;
17      FinMientras
18      Mientras (filas < alto) Hacer
19         Mientras (columnas < ancho) Hacer
20             Escribir Sin Saltar "*";
21             columnas = columnas + 1;
22         FinMientras
23         Escribir ""; // Deja un salto de línea
24         filas = filas + 1;
25         columnas = 0; // Reinicia el contador de columnas
26      FinMientras
27      FinAlgoritmo

```

Código 44: Algoritmo que dibuja un rectángulo de asteriscos en pantalla.

El diagrama de flujo queda de la siguiente manera:

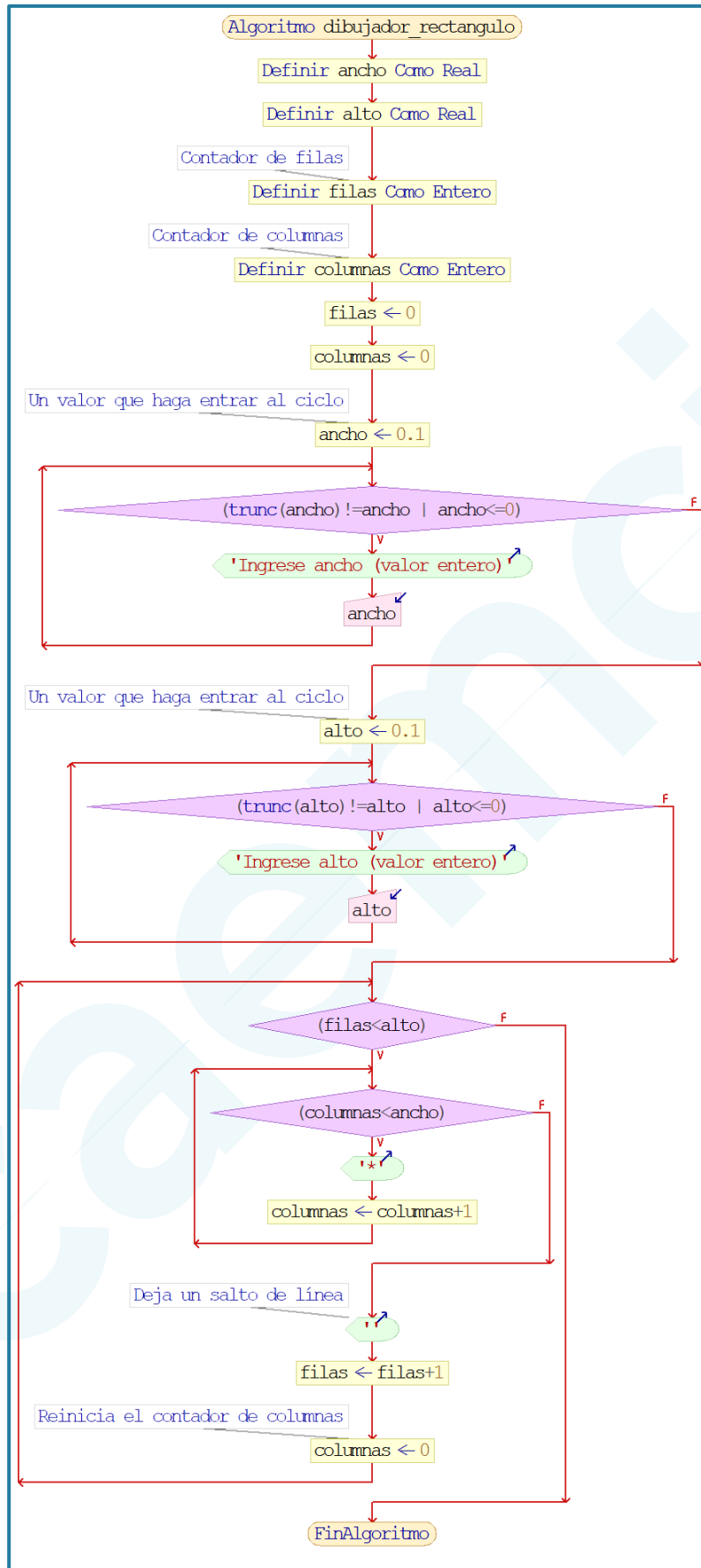


Ilustración 34: Algoritmo que dibuja un rectángulo de asteriscos en pantalla.

La condición:

```
(trunc(ancho) != ancho | ancho <= 0)
```

merece un análisis. Como el operador actuante es el `|`, la manera de que la condición resulte **FALSO** y por ende se salga del ciclo es que ambas premisas resulten **FALSO**. La primera premisa:

```
trunc(ancho) != ancho
```

permite saber si el número ingresado es o no entero. Suponiendo que el usuario ingresara un valor entero, como el `5`, el valor truncado resultaría exactamente igual que el valor sin truncar. En tal caso, la premisa resultaría **FALSO** (se compara si son distintos). Si se ingresara en cambio, por ejemplo, un valor `4.3`, el valor truncado valdría `4`, que no es igual que el valor `4.3` sin truncar. La premisa resultaría **VERDADERO** y el ciclo volvería a iterar.

La premisa:

```
ancho <= 0
```

evalúa si el número ingresado es positivo. Para que resulte **FALSO**, se debe ingresar un número mayor a `0`. La unión de ambas premisas con el operador `|`, permite validar perfectamente lo que se espera: para salir del ciclo de lecturas se debe tener un número entero positivo. Utilizo el mismo razonamiento debajo para pedir el alto.

Respecto a cómo dibujar el rectángulo de asteriscos, utilizo dos ciclos controlados por contador, uno dentro de otro. El ciclo de "más afuera" es quien itera las filas. Por cada iteración de fila, se itera una cantidad de veces, dada por la variable `ancho`, para las columnas. Cuando se terminan de dibujar los asteriscos correspondientes a la fila actual, se escribe un salto de línea (de lo contrario todos los asteriscos se dibujarían en un solo renglón), se incrementa el contador de filas y se reinicia el contador de columnas, para que, en la próxima iteración de las filas, se puedan redibujar las columnas.

La siguiente prueba de escritorio puede ayudarte a aclarar el flujo de las instrucciones. Suponiendo que el usuario ingresa un `ancho` de `4` y un `alto` de `3`, los valores de las variables valdrían lo siguiente:

filas	alto	filas < alto	columnas	ancho	columnas < ancho
0	3	VERDADERO	0	4	VERDADERO
0	3	(No se evalúa)	1	4	VERDADERO
0	3	(No se evalúa)	2	4	VERDADERO
0	3	(No se evalúa)	3	4	VERDADERO

0	3	(No se evalúa)	4	4	FALSO
1	3	VERDADERO	0	4	VERDADERO
1	3	(No se evalúa)	1	4	VERDADERO
1	3	(No se evalúa)	2	4	VERDADERO
1	3	(No se evalúa)	3	4	VERDADERO
1	3	(No se evalúa)	4	4	FALSO
2	3	VERDADERO	0	4	VERDADERO
2	3	(No se evalúa)	1	4	VERDADERO
2	3	(No se evalúa)	2	4	VERDADERO
2	3	(No se evalúa)	3	4	VERDADERO
2	3	(No se evalúa)	4	4	FALSO
3	3	FALSO	0	4	(No se evalúa)

Integrando los conceptos

A continuación, te presento un programa que integra todos los conceptos vistos hasta aquí. Se trata de un algoritmo que pide al usuario un número y la computadora indica si se trata o no de un número primo.

Recordá que un número primo es un número natural mayor que 1 que tiene únicamente dos divisores distintos: él mismo y el 1. De lo contrario, se trata de un número compuesto.

La estrategia del algoritmo es realizar un ciclo que itere y utilice el contador como divisor. En este ejercicio, el corte del ciclo puede darse por **contador** (se llegó hasta el número sin encontrar divisores, por lo tanto, **es primo**) o por **bandera** (se encontró un divisor, por lo tanto, el número **no es primo**).

El diagrama de flujo:

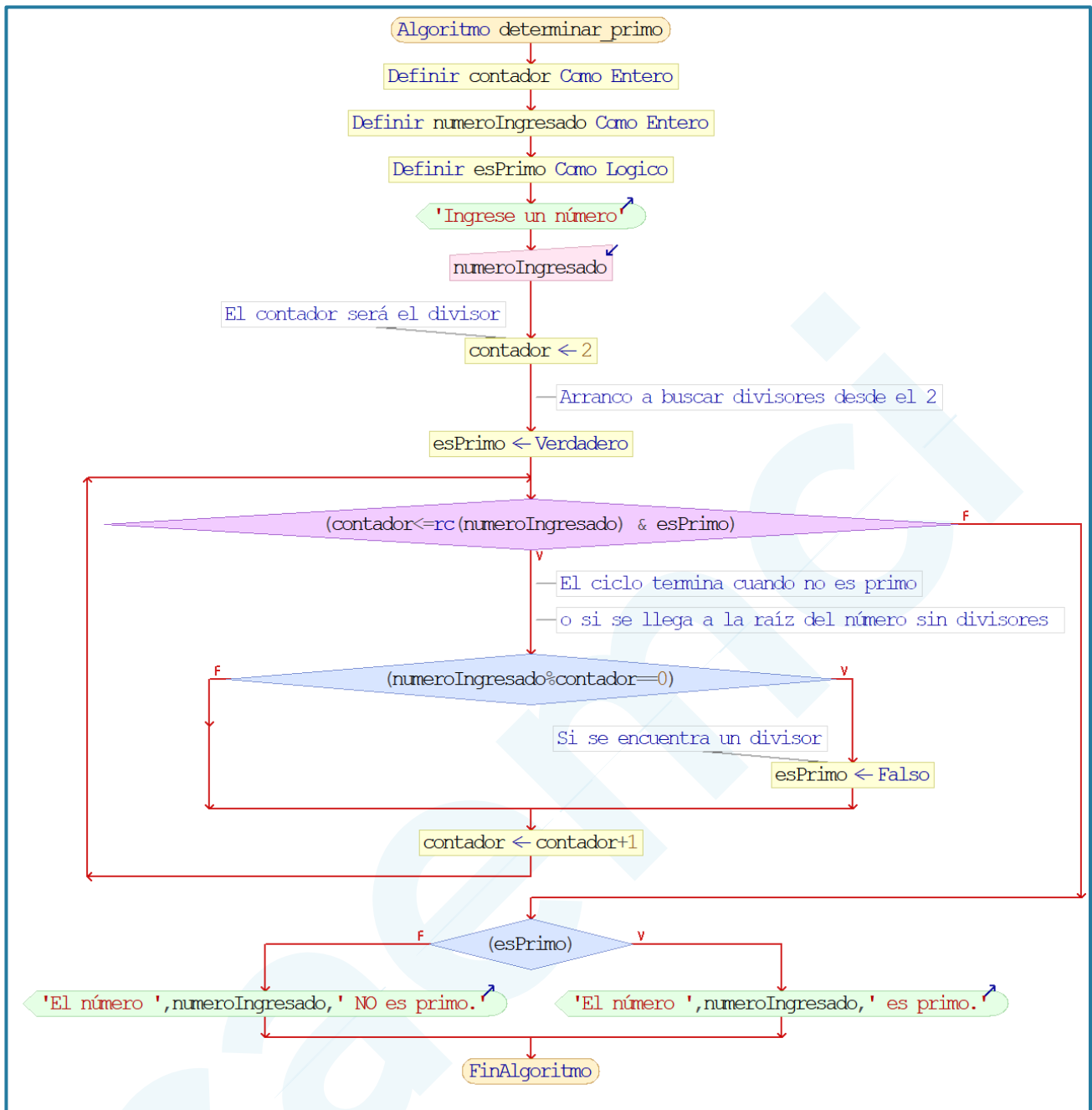


Ilustración 35: Algoritmo que determina si un número es o no primo.

El código:

```

1  Algoritmo determinar_primo
2      Definir contador Como Entero;
3      Definir numeroIngresado Como Entero;
4      Definir esPrimo Como Logico;
5      Escribir "Ingrese un número";

```

```
6      Leer numeroIngresado;
7      contador = 2; // El contador será el divisor
8      // Arranco a buscar divisores desde el 2
9      esPrimo = Verdadero;
10     Mientras (contador <= rc(numeroIngresado) & esPrimo) Hacer
11     // El ciclo termina cuando no es primo
12     // o si se llega a la raíz del número sin divisores
13     Si (numeroIngresado % contador == 0) Entonces
14         esPrimo = Falso; // Si se encuentra un divisor
15     FinSi
16     contador = contador + 1;
17 FinMientras
18 Si (esPrimo) Entonces
19     Escribir "El número ",numeroIngresado," es primo.";
20 Sino
21     Escribir "El número ",numeroIngresado," NO es primo.";
22 FinSi
23 FinAlgoritmo
```

Código 45: Algoritmo que determina si un número es o no primo.

Ejemplo integrador final

Has llegado a la parte final. Ante todo, merecés mis felicitaciones.

La idea para concluir estos conceptos es crear un pequeño juego al que yo llamo "Mayor-Menor-Igual".

Enunciado

"Mayor-Menor-Igual" es el típico juego con barajas donde se predice cómo será el número de la próxima carta que está por salir con respecto a la que se tiene visible actualmente.

El programa comenzará mostrando la explicación del juego y quedará esperando que el usuario presione cualquier tecla para empezar a jugar.

Una vez comenzado el juego, la computadora deberá "pensar" un número al azar entre **1** y **12** (como la baraja española) e informarlo.

El usuario deberá predecir cómo será el próximo número que salga: **MAYOR**, **MENOR** o **IGUAL**. Cualquier opción incorrecta que el usuario ingrese, debe hacer que se repita la pregunta: **MAYOR**, **MENOR** o **IGUAL**.

Mientras el usuario acierte sus predicciones, el juego seguirá la misma temática de "pensar" un número y adivinar cómo será el próximo. Cada acierto valdrá un punto, los cuales se acumularán para luego mostrar el resultado final.

El juego termina cuando no se ha acertado el pronóstico. La computadora muestra el puntaje obtenido (número de aciertos).

Tips

Pausar el programa

En PSeInt existe una instrucción llamada **Esperar Tecla**. La misma permite que la ejecución se pause hasta que se detecte el ingreso de una tecla por parte del usuario.

Usaré esta instrucción para que el usuario tenga el suficiente tiempo de leer las reglas del juego. Él determinará mediante una tecla cuándo empezar a jugar.

Limpiar la pantalla

En PSeInt existe una instrucción llamada **Limpiar Pantalla**. La misma permite que se borre todo lo escrito hasta el momento en la consola.

Usaré esta instrucción para que, por cada nueva apuesta del usuario, se pise a lo anterior, dando la sensación de que las salidas del programa se mantienen en el mismo

lugar de la consola, y no de manera "listada" el usuario tenga el suficiente tiempo de leer las reglas del juego. Él determinará mediante una tecla cuándo empezar a jugar.

Un uso no adecuado de la instrucción **Limpiar Pantalla** podría hacer que se borren las salidas de forma instantánea sin que el usuario vea nada. Generalmente se la utiliza justo antes de una entrada, para que el usuario tenga pausado el programa hasta su ingreso de datos, permitiendo leer la salida o justo antes de una instrucción **Esperar Tecla**.

El código

```

1  Algoritmo mayor_menor_igual
2      Definir cartaActual Como Entero;
3      Definir proxCarta Como Entero;
4      Definir opc Como Entero;
5      Definir puntaje Como Entero;
6      Definir opcionValida Como Logico;
7      Definir fueDerrotado Como Logico;
8      Escribir "Bienvenido a MAYOR-MAYOR-IGUAL";
9      Escribir "La computadora generará un número al azar entre 1 y 12.";
10     Escribir "Deberás predecir si el próximo será MAYOR, MENOR o IGUAL.";
11     Escribir "Cada acierto vale 1 punto. El juego termina cuando no adivinás.";
12     Escribir "";
13     Escribir "Presioná cualquier tecla para continuar...";
14     Esperar Tecla;
15     Limpiar Pantalla;
16     fueDerrotado = Falso; // Inicialización de la bandera
17     puntaje = 0; // Inicialización del puntaje
18     cartaActual = azar(12) + 1; // Genero la primera carta
19     Mientras (!fueDerrotado) Hacer // Termina cuando sea derrotado
20         opcionValida = Falso; // Inicialización de la bandera
21         proxCarta = azar(12) + 1; //Se "piensa" la carta que saldrá
22         Mientras (!opcionValida) Hacer //Termina si la opción es válida
23             Escribir "Salió el [" , cartaActual , "];

```



```

24      Escribir ""; // Línea en blanco
25      Escribir "La próxima carta será:";
26      Escribir ""; // Línea en blanco
27      Escribir "[1] MAYOR";
28      Escribir "[2] MENOR";
29      Escribir "[3] IGUAL";
30      Escribir ""; // Línea en blanco
31      Escribir "Escribí la opción correspondiente.";
32      Leer opc;
33      Escribir ""; // Línea en blanco
34      Segun (opc) Hacer
35          1: // MAYOR
36              Si (proxCarta > cartaActual) Entonces
37                  Escribir "Acertaste. Salió el ",proxCarta;
38                  puntaje = puntaje + 1; // Suma un punto
39              Sino
40                  Escribir "Perdiste. Salió el ",proxCarta;
41                  fueDerrotado = Verdadero; // ¡Bandera!
42              FinSi
43              opcionValida = Verdadero; // ¡Bandera!
44          2: // MENOR
45              Si (proxCarta < cartaActual) Entonces
46                  Escribir "Acertaste. Salió el ",proxCarta;
47                  puntaje = puntaje + 1; // Suma un punto
48              Sino
49                  Escribir "Perdiste. Salió el ",proxCarta;
50                  fueDerrotado = Verdadero; // ¡Bandera!
51              FinSi
52              opcionValida = Verdadero; // ¡Bandera!
53          3: // IGUAL
54              Si (proxCarta == cartaActual) Entonces
55                  Escribir "Acertaste. Salió el ",proxCarta;

```

```
56         puntaje = puntaje + 1; // Suma un punto
57     Sino
58         Escribir "Perdiste. Salió el ",proxCarta;
59         fueDerrotado = Verdadero; // ¡Bandera!
60     FinSi
61     opcionValida = Verdadero; // ¡Bandera!
62     De Otro Modo:
63         Escribir "Opción inválida";
64     FinSegun
65     Escribir ""; // Línea en blanco
66     FinMientras
67     Escribir "Presioná una tecla para continuar...";
68     Esperar Tecla;
69     Limpiar Pantalla;
70     cartaActual = proxCarta; // La carta que salió ahora es la actual
71     FinMientras
72     Escribir "Tu puntuación: " , puntaje;
73     FinAlgoritmo
```

Código 46: Algoritmo MAYOR-MENOR-IGUAL.

Tabla de códigos

Código 1: Cómo preparar café.....	15
Código 2: Algoritmo con instrucción de salida.....	24
Código 3: Algoritmo con dos instrucciones de salida.....	25
Código 4: Algoritmo con salida sin salto de línea.....	25
Código 5: Algoritmo con salida de expresiones numéricas.....	26
Código 6: Diferencia entre datos numéricos y alfanúmericos.....	26
Código 7: Definición de variables.....	34
Código 8: Algoritmo que carga y muestra variables.....	36
Código 9: Algoritmo que modifica el valor de una variable.....	38
Código 10: Algoritmo que diferencia entre un identificador de variable y una cadena....	38
Código 11: Algoritmo que actualiza una variable.....	39
Código 12: Algoritmo que suma dos enteros provistos por el usuario.....	40
Código 13: Algoritmo que suma dos enteros y utiliza el operador de concatenación. .	41
Código 14: Algoritmo que calcula la raíz cuadrada de un número usando funciones. ..	42
Código 15: Algoritmo con comentarios	46
Código 16: Ejemplo integrador de los conceptos de flujo secuencial.....	47
Código 17: Verificación de expresiones booleanas con operadores relacionales entre números.....	51
Código 18: Verificación de expresiones booleanas con operadores relacionales entre cadenas.....	52
Código 19: Algoritmo que valida un pase según la edad.....	53
Código 20: Algoritmo que valida o no un pase según la edad.....	57
Código 21: Algoritmo que reconoce el signo de un número.....	60
Código 22: Primera parte del algoritmo reconocedor del momento del día.....	61
Código 23: Segunda parte del algoritmo reconocedor del momento del día.....	63
Código 24: Demostración del operador NOT.....	64
Código 25: Algoritmo que valida un pase según ciertas condiciones con operador el OR.....	65
Código 26: Algoritmo que valida una opción.....	66
Código 27: Algoritmo alternativo que valida una opción.....	66

Código 28: Algoritmo que valida un pase según ciertas condiciones con el operador AND.....	67
Código 29: Algoritmo que valida que un número esté entre 0 y 10.....	68
Código 30: Algoritmo reconocedor del momento del día optimizado.....	69
Código 31: Algoritmo que simula un menú telefónico con una estructura de selección múltiple Según . . . FinSegun.....	73
Código 32: Algoritmo que simula un menú telefónico contemplando opciones incorrectas con una estructura de selección múltiple Según . . . FinSegun.....	75
Código 33: Algoritmo que simula una calculadora básica.	77
Código 34: Algoritmo que muestra cinco veces una cadena.	80
Código 35: Algoritmo que muestra cinco veces una cadena contando hacia adelante.....	82
Código 36: Algoritmo que muestra cinco veces una cadena contando hacia atrás.....	83
Código 37: Algoritmo que muestra los primeros 20 números pares, aclarando cuáles son múltiplos de 6.	84
Código 38: Algoritmo que calcula la sumatoria de los primeros 10 números naturales.	86
Código 39: Algoritmo que calcula la sumatoria de los números ingresados por el usuario.....	87
Código 40: Algoritmo que calcula la sumatoria de los primeros 10 números naturales.	88
Código 41: Algoritmo que calcula la sumatoria de números ingresados por el usuario utilizando ciclo controlado por bandera.....	90
Código 42: Algoritmo alternativo que calcula la sumatoria de números ingresados por el usuario utilizando ciclo controlado por bandera.	92
Código 43: Algoritmo que calcula la sumatoria de números ingresados por el usuario con validación de opciones.	94
Código 44: Algoritmo que dibuja un rectángulo de asteriscos en pantalla.	95
Código 45: Algoritmo que determina si un número es o no primo.....	100
Código 46: Algoritmo MAYOR-MENOR-IGUAL.	104

Tabla de ilustraciones

Ilustración 1: Componentes de un algoritmo.	12
Ilustración 2: Diagrama de flujo de cómo preparar café	14
Ilustración 3: Ventana inicial del instalador.	16
Ilustración 4: Acuerdo de licencia de PSeInt.	17
Ilustración 5: Directorio de instalación de PSeInt.	17
Ilustración 6: Ventana final del instalador de PSeInt.	18
Ilustración 7: Menú Configurar.	18
Ilustración 8: Parámetros del lenguaje usado en este libro.	19
Ilustración 9: Editando el primer programa.	20
Ilustración 10: Ejecución del primer programa.	20
Ilustración 11: Flujo secuencial.	23
Ilustración 12: Tabla de caracteres ASCII.	28
Ilustración 13: Simulación de memoria con variables sin inicializar.	33
Ilustración 14: Simulación de memoria con variables inicializadas.	37
Ilustración 15: Ejemplo integrador de los conceptos de flujo secuencial.	48
Ilustración 16: Flujo de selección simple.	52
Ilustración 17: Algoritmo que valida un pase según la edad.	54
Ilustración 18: Flujo de selección doble.	55
Ilustración 19: Algoritmo que valida o no un pase según la edad.	57
Ilustración 20: Anidamiento de estructuras de selección.	58
Ilustración 21: Algoritmo que reconoce el signo de un número.	59
Ilustración 22: Primera parte del reconocedor del momento del día.	61
Ilustración 23: Segunda parte del reconocedor del momento del día.	61
Ilustración 24: Flujo de selección múltiple.	71
Ilustración 25: Algoritmo que simula un menú telefónico con una estructura de selección múltiple Según . . . FinSegun.	74
Ilustración 26: Flujo de repetición.	79
Ilustración 27: Algoritmo que muestra cinco veces una cadena.	80
Ilustración 28: Algoritmo que muestra cinco veces una cadena contando hacia atrás.	83
Ilustración 29: Algoritmo que muestra los primeros 20 números pares, aclarando cuáles son múltiplos de 6.	84

Ilustración 30: Algoritmo que calcula la sumatoria de los primeros 10 números naturales.....	85
Ilustración 31: Algoritmo que calcula la sumatoria de los números ingresados por el usuario.....	87
Ilustración 32: Algoritmo que calcula la sumatoria de números ingresados por el usuario utilizando ciclo controlado por bandera.	91
Ilustración 33: Algoritmo que calcula la sumatoria de números con validación de opción.	93
Ilustración 34: Algoritmo que dibuja un rectángulo de asteriscos en pantalla.	96
Ilustración 35: Algoritmo que determina si un número es o no primo.	99