Elementos básicos en programación

- Salidas
- Entradas
- Almacenamiento
- Procesamiento
- Funciones
- Permitidme un comentario...

Salidas

Normalmente programamos porque queremos obtener un resultado. Ese resultado es lo que llamamos habitualmente **SALIDA** del programa, si bien todo aquello que el programa muestra a la persona usuaria recibe la misma denominación.

Cuando se están dando los primeros pasos en programación, se suele comenzar por la creación del programa **Hola Mundo** que no es más que un programa que muestra ese mensaje en pantalla. Este sencillo programa nos ayudará a familiarizarnos con las salidas. Veamos cómo hacerlo paso a paso:

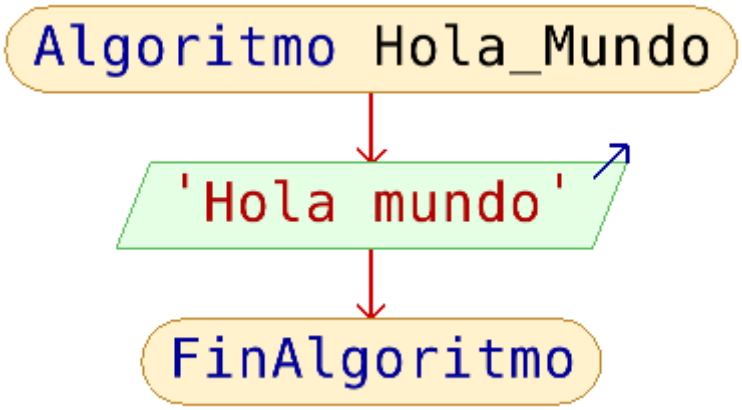
Pasos 1 y 2: Análisis y diagrama de flujo del programa *Hola mundo*

Según lo visto en el primer apartado tendría los siguientes elementos:

- Inicio y fin del algoritmo.

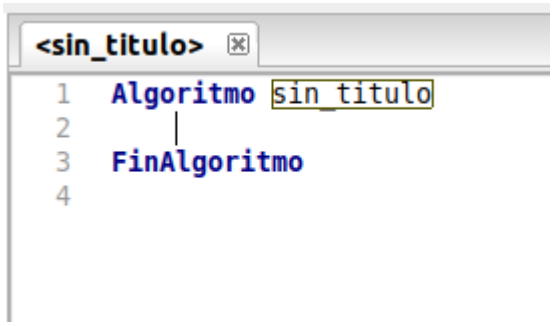
- Una única salida que mostraría ese mensaje.

SOLUCIÓN:

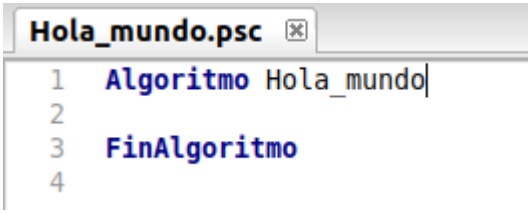


Pasos 3, 4 y 5: Codificación, compilación y verificación del programa *Hola mundo* con PSeInt

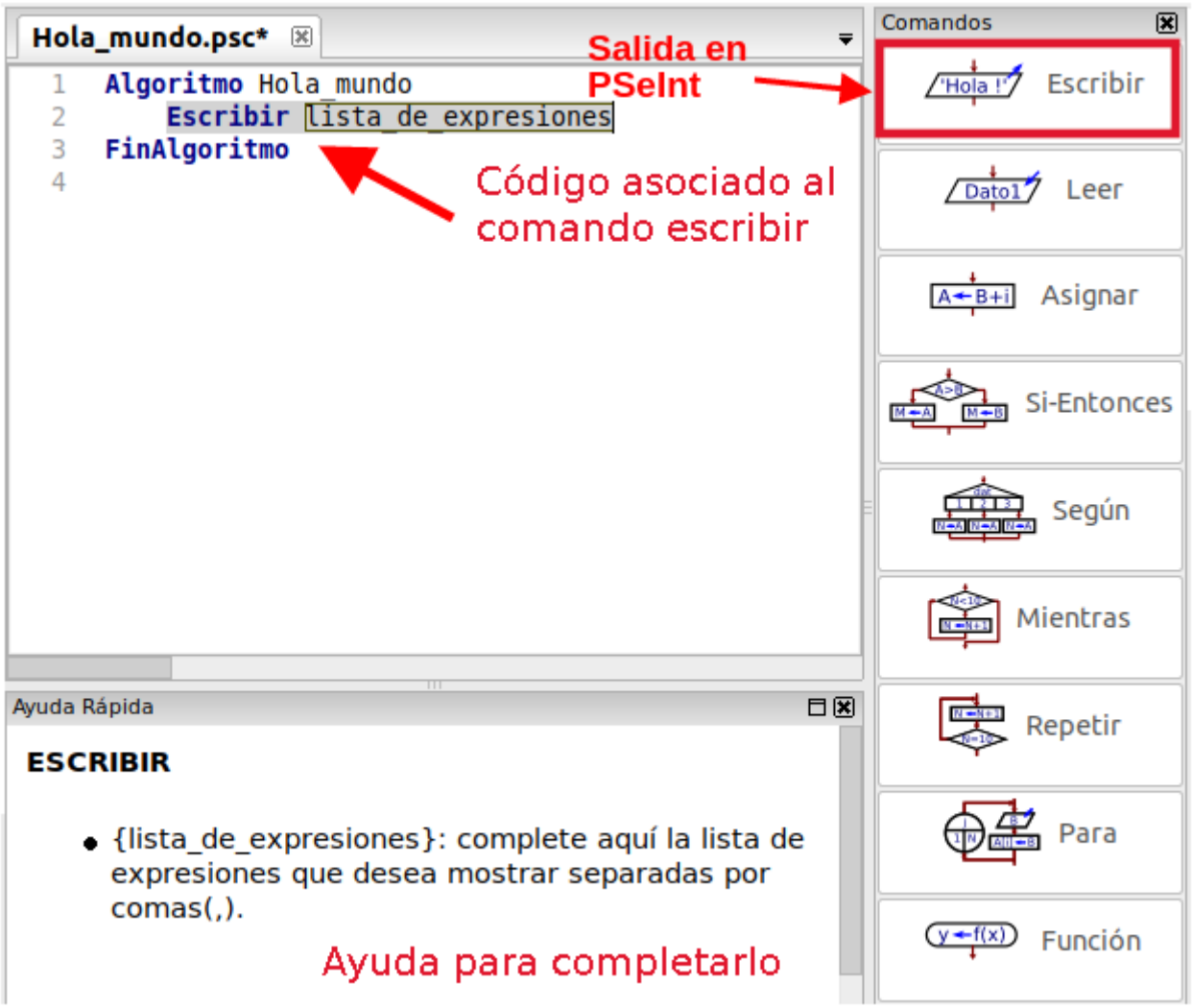
PSeInt ya arranca por defecto con el inicio y final del algoritmo escrito:



En primer lugar le cambiaríamos el nombre al algoritmo



y por último debemos encontrar el comando para mostrar una **salida** en pantalla. Eso lo realiza el comando **Escribir.**

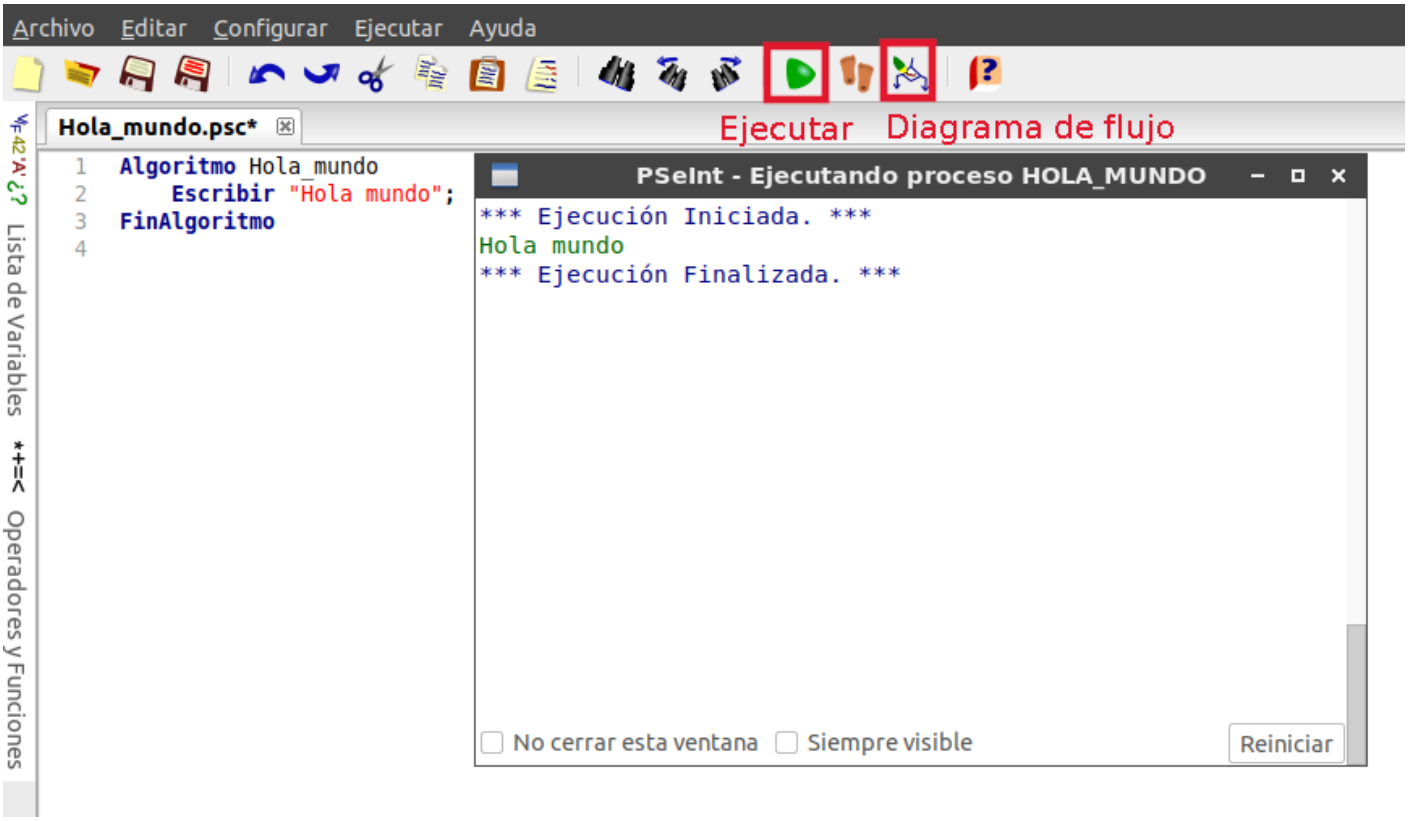


Si hacemos clic sobre **Escribir**, se añade automáticamente el código en la línea del programa donde tengamos el cursor, con la sintaxis correcta, y nos muestra en la parte inferior una ayuda para completar la sintaxis.

En este caso, al ser una frase, deberemos escribir el **texto que queremos mostrar entrecomillado**. También es conveniente terminar la línea con un **punto y coma** para acostumbrarnos puesto que así es en la mayor parte de los lenguajes de programación, si bien PSeInt al ser pseudocódigo no es tan estricto con este tipo de errores y nos ejecutará el programa igualmente.

```
Hola_mundo.psc*
1  Algoritmo Hola_mundo
2      Escribir "Hola mundo";
3  FinAlgoritmo
4
```

Si ahora queremos ver el correcto funcionamiento del programa, haremos clic en la herramienta **Ejecutar** (el triángulo verde) y se nos abrirá la ventana de ejecución del programa o consola donde veremos el resultado. Esta instrucción realiza la compilación y la verificación en el mismo paso. En programas más complejos podremos utilizar la herramienta **Ejecutar paso a paso,** muy útil para la depuración de errores.



Si queremos comprobar que nuestro diagrama de flujo es correcto, simplemente deberemos hacer clic en la herramienta **Dibujar diagrama de flujo** y veremos que es el mismo que habíamos dibujado al principio.

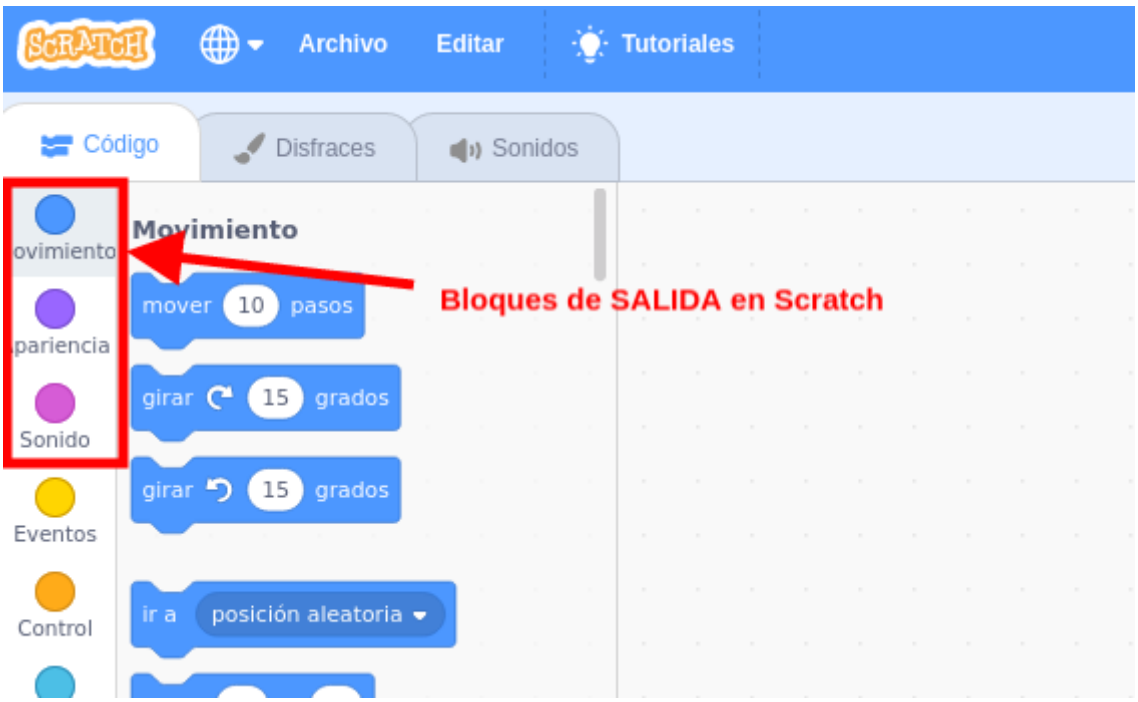
Como se puede comprobar PSeInt es un programa que proporciona salidas muy simples de tipo alfanumérico. No obstante, su sencillez nos ayuda a comprender el funcionamiento básico. Ahora realizaremos la comparación con Scratch, un programa cuya potencia gráfica es muchísimo mayor.

Pasos 3, 4 y 5: **Codificación, compilación y verificación del programa *Hola mundo* con Scratch.**

En primer lugar remarcar que Scratch es un programa muchísimo más complejo y completo que PSeInt. Para empezar en Scratch nuestros programas tienen varios objetos ejecutando código a la vez, y las **Salidas** que proporciona son mucho más complejas y sofisticadas.

Los bloques de código asociados a las salidas son:

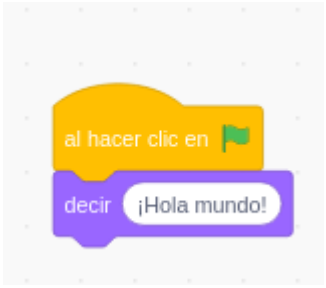
- **Movimiento**: producen un movimiento en el objeto seleccionado.
- **Apariencia**: producen algún cambio en la apariencia del objeto seleccionado.
- **Sonido**: producen algún sonido asociado al objeto seleccionado.



En concreto para nuestro programa nos van a interesar un par de bloques de los de **Apariencia**:

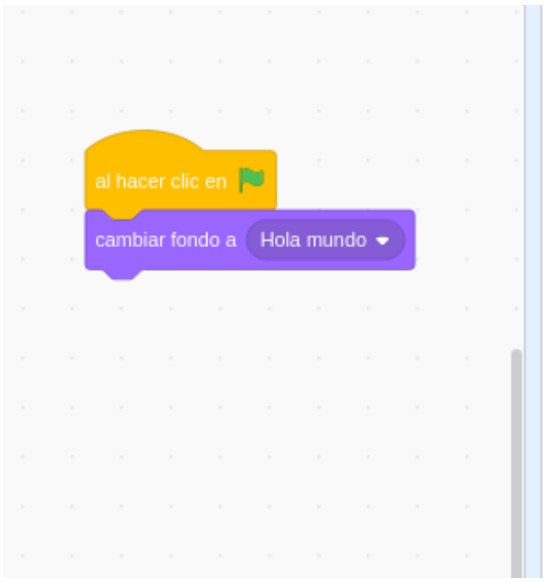
Vemos que dichos bloques mostrarán al objeto diciendo un mensaje, en el primer caso durante un tiempo determinado, en el segundo de forma indefinida.

Por último nos quedará marcar el inicio y final del programa. En Scratch el inicio se marca haciendo clic sobre la bandera verde que hay en la parte superior de la ventana de programa, y esa orden se encuentra en el bloque **Eventos**. El final no se especifica.



Al hacer clic en la bandera verde, el objeto al que le hemos asignado el código saldrá diciendo ese mensaje.

En este caso hemos asociado la SALIDA a un objeto/personaje. De igual forma se lo podíamos haber asociado al escenario, simplemente cambiando el fondo a uno que incluyera ese mensaje, en el bloque Apariencia, y que previamente tendríamos que haber creado.



Financiado por el Ministerio de Educación y Formación Profesional y por la Unión Europea - NextGenerationEU

Financiado por la Unión Europea NextGenerationEU
Plan de Recuperación, Transformación y Resiliencia
GOBIERNO DE ARAGON

Entradas

Un programa como el ejemplo del programa *Hola mundo* visto en el apartado anterior no es lo habitual en programación. Cada vez más, buscamos realizar programas que realicen acciones no meramente preprogramadas y automatizadas, sino que reaccionen en función de unos parámetros suministrados por la persona usuaria. Todos aquellos datos que se le proporcionan al programa de forma externa para su posterior procesamiento es lo que llamamos **ENTRADAS**.

Las entradas pueden ser suministradas de dos formas:

- De forma **manual**: se le solicitan a la persona usuaria y esta los introduce, normalmente vía ratón o teclado.
- De forma **automatizada**: mediante sensores que facilitan dicha información del entorno. Por supuesto con PSeInt no podemos hacer esto, pero con Scratch sí.

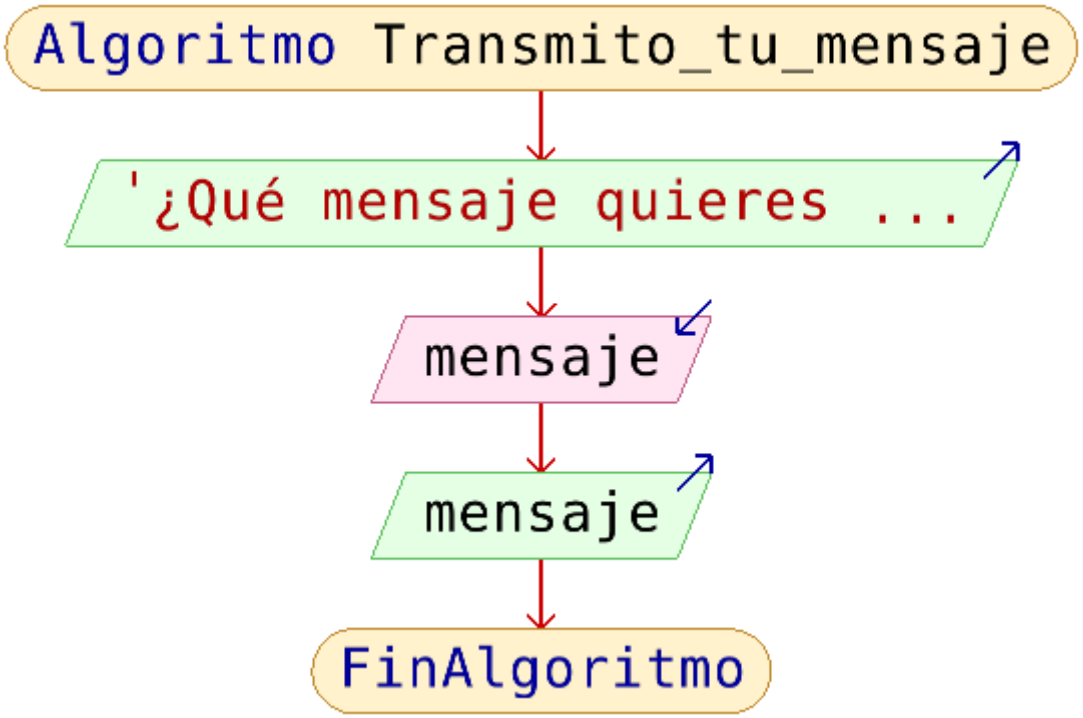
Para entender mejor el concepto de **Entrada** realizaremos una variante del programa **Hola mundo** que llamaremos **Transmito tu mensaje** en el que el programa solicitará de la persona usuaria el mensaje que quiere transmitir, y a continuación lo mostrará en pantalla. Vayamos paso a paso:

Pasos 1 y 2: Análisis y diagrama de flujo del programa *Transmito tu mensaje*

. Veamos en primer lugar los elementos necesarios en dicho programa y su diagrama de flujo:

- **Inicio y fin** de programa.
- **Salida**: Se necesita una primera salida preguntando por el contenido del mensaje y una segunda mostrándolo.
- **Entrada**: el contenido del mensaje.
- En este programa aparece un nuevo concepto que es el de **variable**, es decir un "contenedor" que almacena los datos que recibe de la entrada, para su posterior procesamiento. Lo veremos con más detalle en el siguiente apartado dedicado a almacenamiento de datos.

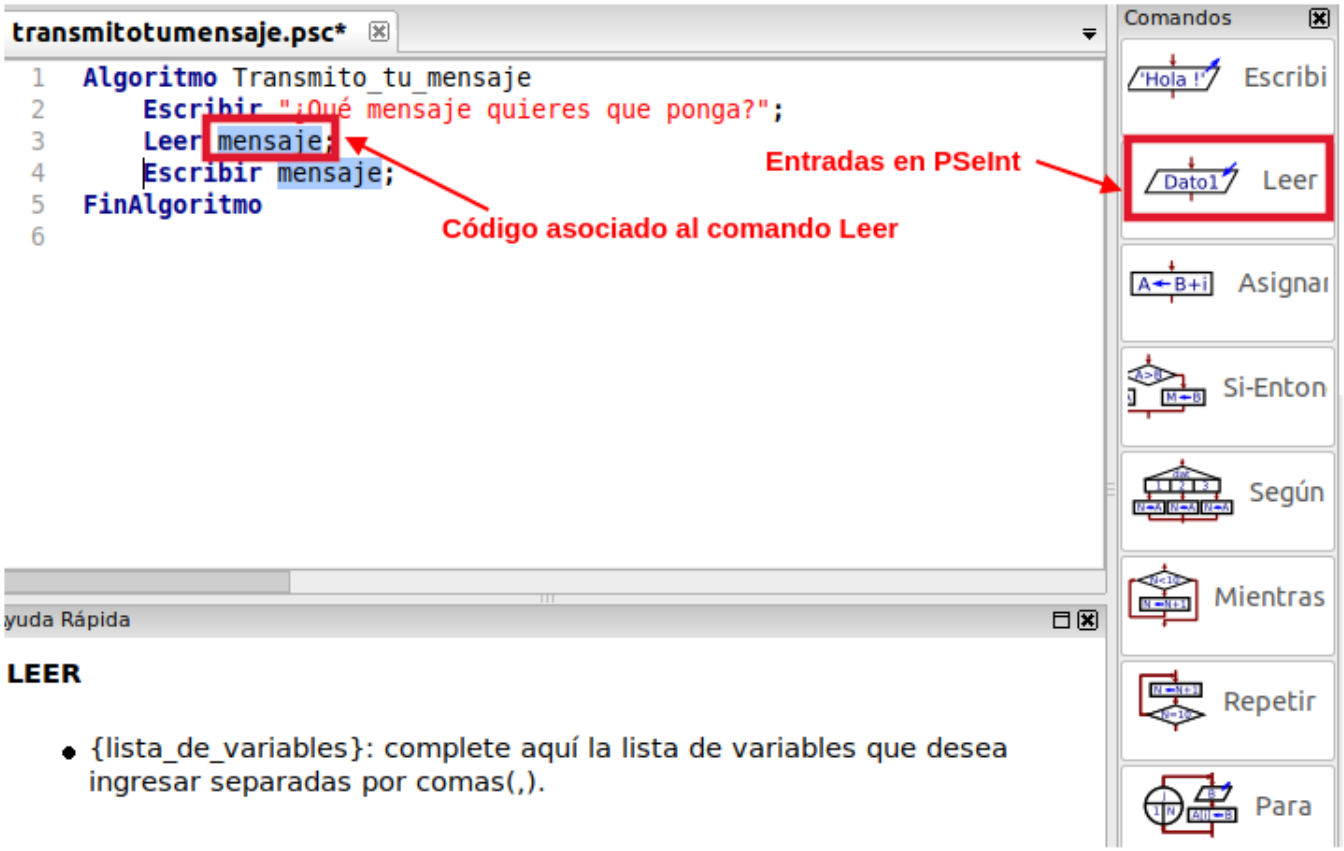
El diagrama de flujo sería:



Pasos 3, 4 y 5: Codificación, compilación y verificación del programa *Transmito tu mensaje* con PSeInt

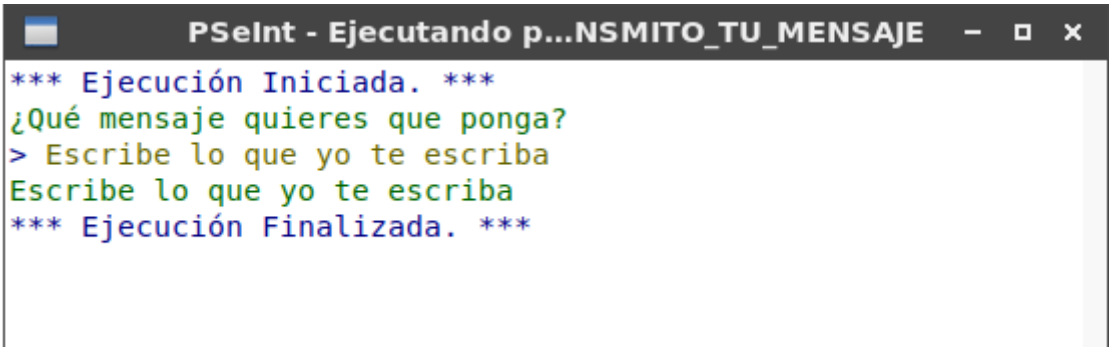
Partiremos de nuestro fichero Hola_mundo.psc, le cambiaremos el nombre al algoritmo y al fichero, y en este caso introduciremos una vez más el comando **Escribir** puesto que necesitamos dos salidas.

Para recibir **entradas** en PSeInt se utiliza el comando **Leer** seguido de la **variable** en la que se almacenará el contenido recibido. Nosotros le llamaremos simplemente *mensaje*.



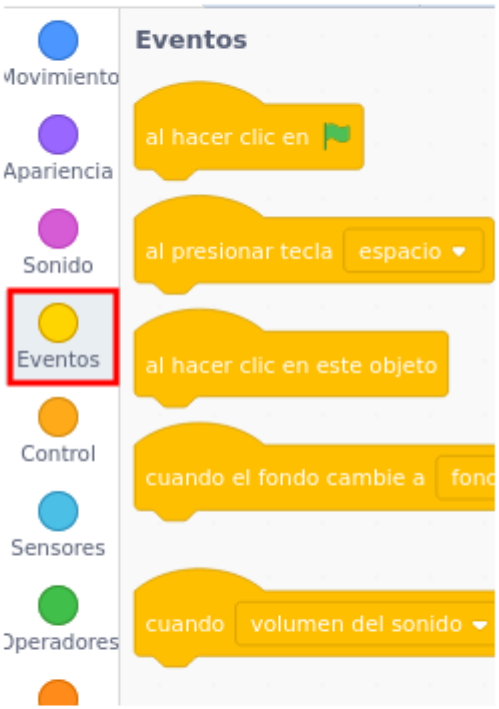
PSeInt nos facilita el no tener que definir el tipo de contenido a recibir previamente (textual, numérico...) sino que se adapta a la respuesta, en este caso una cadena de caracteres.

Para comprobar su funcionamiento, haremos clic en el botón **Ejecutar** de la barra de herramientas y comprobaremos que realiza lo esperado.



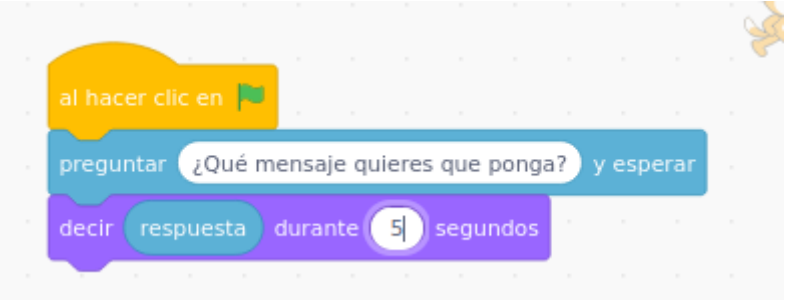
Pasos 3, 4 y 5: Codificación, compilación y verificación del programa *Transmito tu mensaje* con Scratch

En Scratch parte de los bloques relacionados con la entrada de datos en el programa se encuentran en el menú **SENSORES**, y tienen color **azul claro**. También proporcionamos entradas con el menú **EVENTOS**, con bloques para indicar qué acción exterior determinará el que el programa se ejecute de una forma u otra.



Como veíamos antes, en *SENSORES* se incluyen tanto bloques para preguntar y recibir respuestas directas de las personas usuarias, como otro tipo de entradas relacionadas con estar tocando algo (otros objetos, colores), estar a determinada distancia, alcanzar un volumen de sonido, encontrarse en una zona determinada del lienzo, pulsar una tecla o el ratón, etc...

Si quisiéramos programar algo similar al programa anterior tendríamos que utilizar los siguientes bloques:



Scratch almacena lo introducido por teclado en una variable predefinida llamada **respuesta,** disponible en el mismo bloque de sensores. Precisamente las variables y otras estructuras de almacenamiento de datos son el objetivo del siguiente punto.

Financiado por el Ministerio de Educación y Formación Profesional y por la Unión Europea - NextGenerationEU

Almacenamiento

Ya hemos visto que para poder operar con los datos, previamente hemos debido reservar unos espacios de memoria en los que almacenarlos. A esto es a lo que llamamos **estructuras de almacenamiento de datos.**

Para que los datos puedan ser correctamente tratados, estas estructuras deben ser definidas mediante:

- Identificador: nombre que le damos al dato durante el programa para referirnos a él.
- Tipo: naturaleza y rango de valores que puede almacenar.
- Valor: contenido concreto del dato en ese momento.

En este curso introductorio vamos a conocer dos estructuras únicamente, variables y vectores (también llamados arrays), si bien solo vamos a deternos en profundidad en la primera.

Variables y constantes

Son espacios que almacenan **un único DATO** de diferente naturaleza, y al que se le asocia un nombre que lo identifica. Los datos que pueden almacenar son:

- Numéricos: pueden ser enteros (int) o reales (float), según si aceptan o no decimales, generalmente con notación anglosajona donde el decimal se marca con un punto.
- Caracteres (char): letras o signos tipográficos. Se definen entrecomillando el carácter con una comilla simple.
- Cadenas de caracteres (string): palabras o frases. Se definen entrecomillando el texto con doble comilla.
- Lógicos (boolean): admite los valores de VERDADERO o FALSO.

Las variables, como su nombre indica, son estructuras de datos que van a tomar diferentes valores a lo largo de la ejecución del programa, mientras que las constantes adoptan el mismo valor durante todo el programa (Por ejemplo el número PI, o el porcentaje del IVA).

Es muy importante seleccionar adecuadamente el nombre de las variables y constantes para que ayuden a la comprensión de los datos almacenados. Generalmente las constantes se definen con nombres en mayúscula y las variables en minúscula.

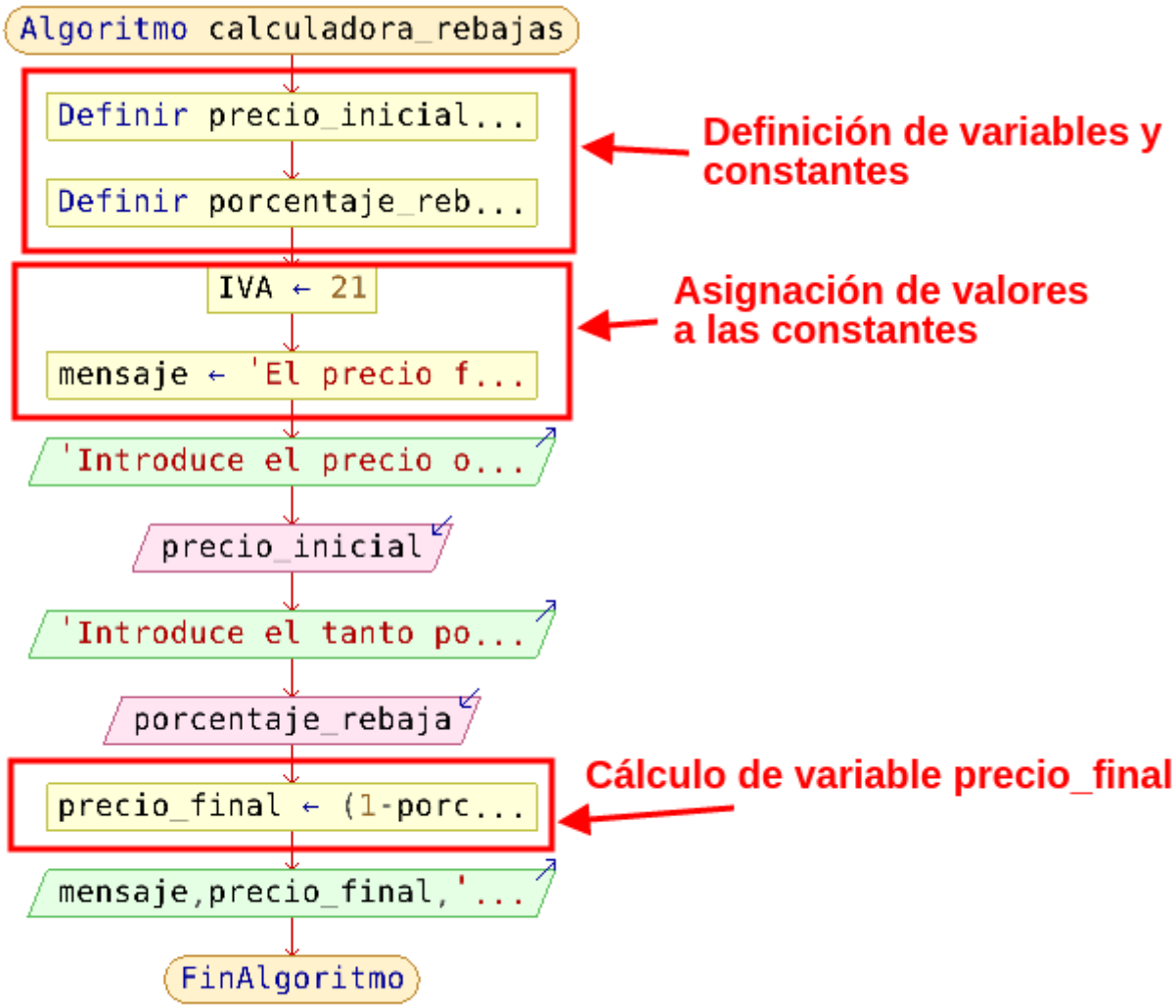
Para conocer más de su funcionamiento, vamos a realizar un pequeño programa que solicite un precio, un porcentaje de oferta y a partir de un porcentaje de IVA (Constante) muestre un mensaje con el precio total a pagar. Le llamaremos CALCULADORA DE REBAJAS.

Pasos 1 y 2: Análisis y diagrama de flujo del programa *Calculadora de Rebajas*

Los elementos implicados serán:

- **Inicio y fin** de programa.
- **Salidas**: Solicitar precio original y porcentaje de oferta, y mostrar el precio final.
- **Entradas**: Precio original, porcentaje de oferta.
- **Almacenamiento**: precio original (número real porque puede ser decimal), porcentaje de oferta (número entero porque usaremos el valor en tanto por ciento), precio final (número real), porcentaje de IVA (constante), y opcionalmente el mensaje a mostrar (constante)
- **Procesamiento**: sumas, multiplicaciones y divisiones.

El diagrama de flujo sería:

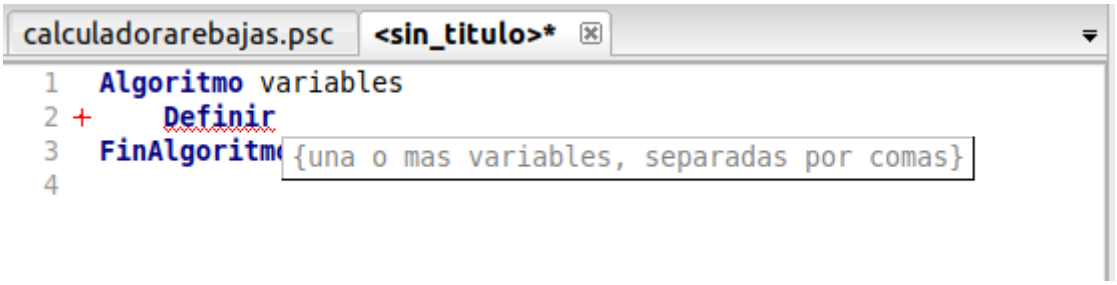


A las variables de igual forma también se les podrían asignar valores iniciales, aunque vayan a ser modificados a lo largo del programa. Si no se hace, asumen como valor inicial el 0 o vacío.

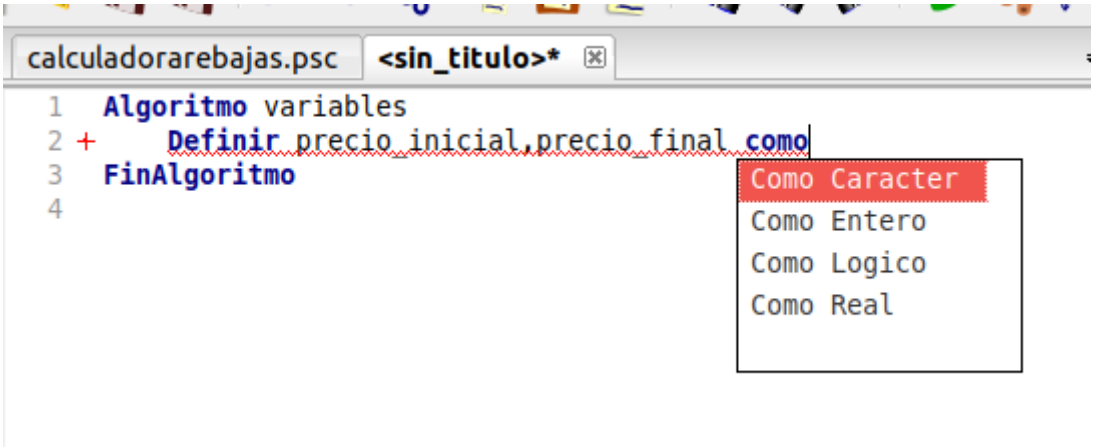
Pasos 3, 4 y 5: Codificación, compilación y verificación del programa *Calculadora de Rebajas* con PSeInt

Como hemos visto en el ejemplo de Entradas, PSeInt es un programa menos "riguroso" con el código, y nos ha permitido utilizar variables sin haberlas definido previamente (como hacíamos con num1 y num2, variables que almacenaban los valores de los números introducidos)

No obstante, el procedimiento ordinario en todo programa es definir previamente las variables (y constantes) que vamos a necesitar, mediante la asignación de un **identificador** y un **tipo de datos** a almacenar. En PSeInt esto se realiza mediante la instrucción **Definir.** Al escribir dicha instrucción, la ayuda nos solicita el nombre de las variables que queramos definir, separadas entre comas.

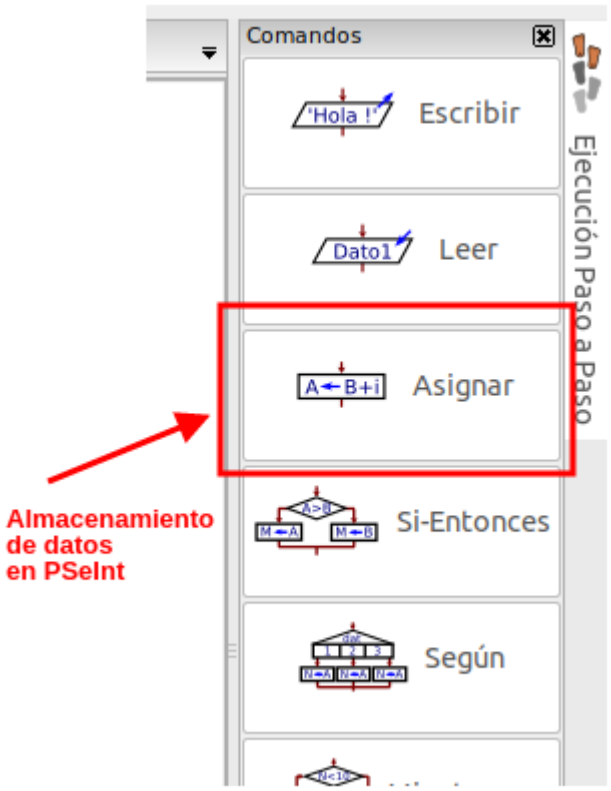


Podemos también especificar el tipo de datos que van a contener, añadiendo después del nombre la instrucción **como,** que nos abrirá las cuatro opciones disponibles en PSeInt: **entero, real, carácter o lógico**, también llamado booleano. En el caso de las cadenas de caracteres, no permite su definición como tal, si bien veremos que mediante asignación podemos almacenarlas en variables de igual forma.



Realizamos este proceso tantas veces como necesitemos, por el tipo de datos que hemos definido en el problema. En el caso de las constantes, les asignaremos su valor (21% en el caso del IVA y el texto del mensaje que queramos)

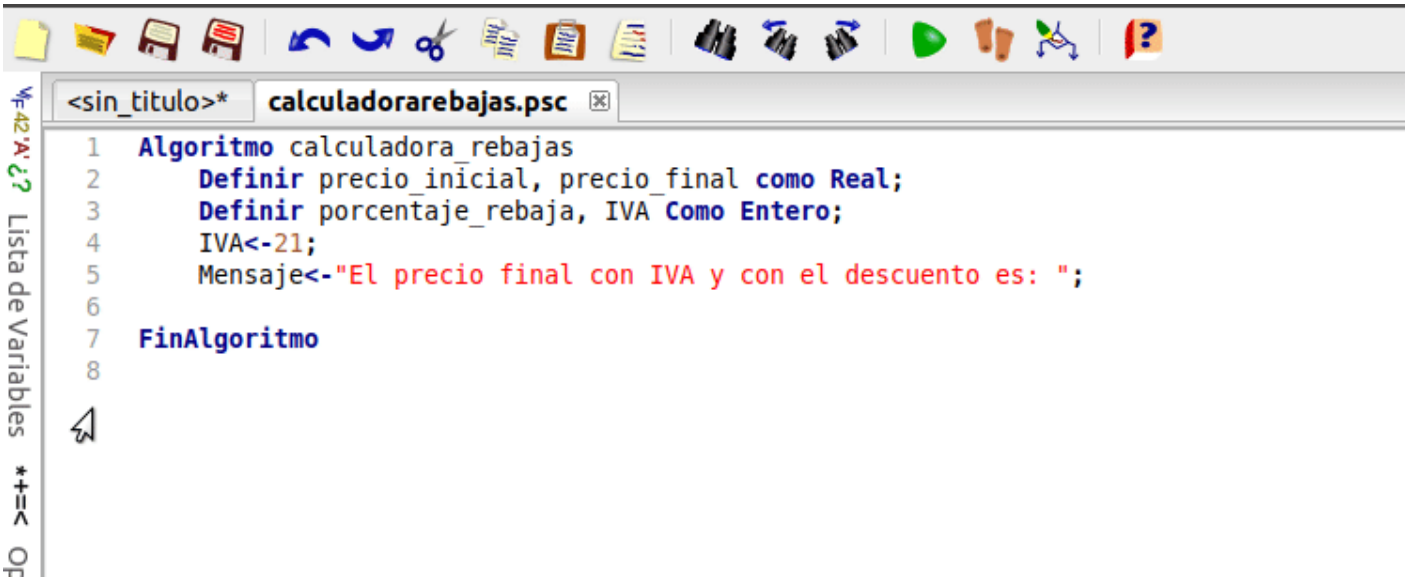
Para almacenar un valor en una variable o constante se utiliza el comando **Asignar**, de la ventana de la derecha.



También se pueden escribir directamente el signo menor y el guión mediante el teclado.

```
Algoritmo calculadora_rebajas
    Definir precio_inicial, precio_final como Real;
    Definir porcentaje_rebaja, IVA Como Entero;
    IVA<-21;
    Mensaje<-"El precio final con IVA y con el descuento es: ";
```

En el momento en que definimos variables y constantes en el programa, si hacemos clic en la pestaña situada a la izquierda llamada **Lista de Variables** se nos abre una ventana con las variables definidas.



Una vez definidas las variables que van a contener la información necesaria, solicitaremos y almacenaremos los valores del precio inicial y el porcentaje de rebaja.

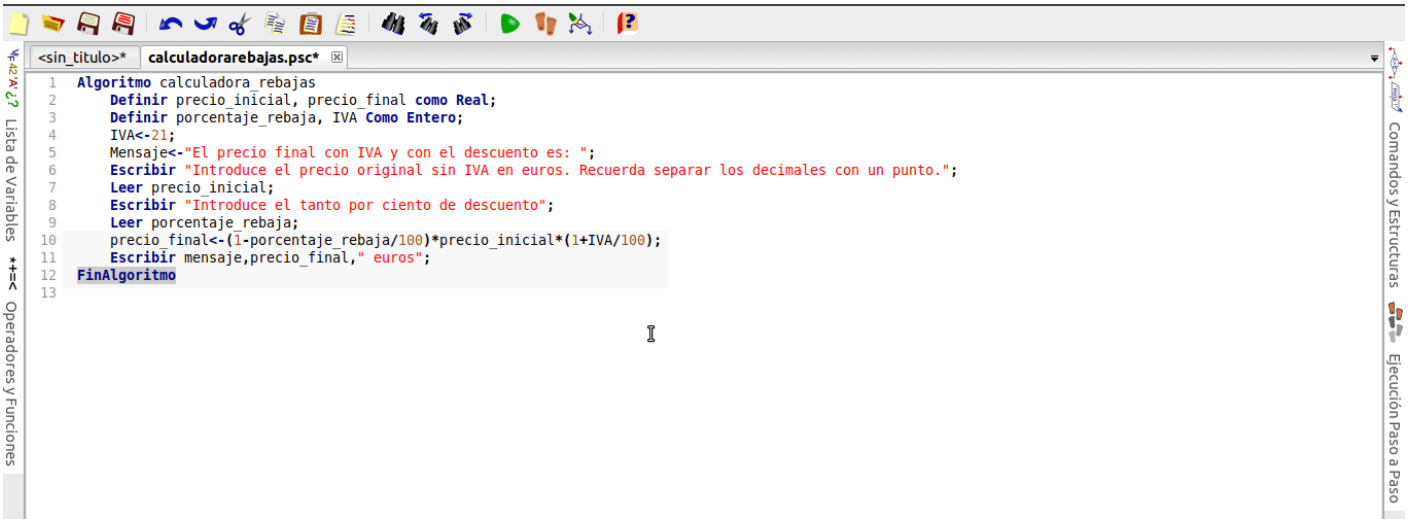
```
Escribir "Introduce el precio original sin IVA en euros. Recuerda separar los decimales con un punto.";
Leer precio_inicial;
Escribir "Introduce el tanto por ciento de descuento";
Leer porcentaje rebaja;
```

Por último, operaremos con ellos mediante operaciones aritméticas (ver siguiente punto) y asignaremos el valor obtenido a la variable precio_final que será la que mostraremos por consola.

```
    precio_final<-(1-porcentaje_rebaja/100)*precio_inicial*(1+IVA/100);
    Escribir mensaje,precio_final," euros";
FinAlgoritmo
```

Como se puede apreciar en la última fila del programa, la instrucción **Escribir** permite la concatenación de diferentes tipos de datos sin más que separarlos entre comas.

Ya solo nos quedará comprobar el correcto funcionamiento del programa dándole a **Ejecutar**. Debido a la mayor longitud de este programa puede ser un buen momento para comprobar como funciona la orden **Ejecutar paso a paso.**



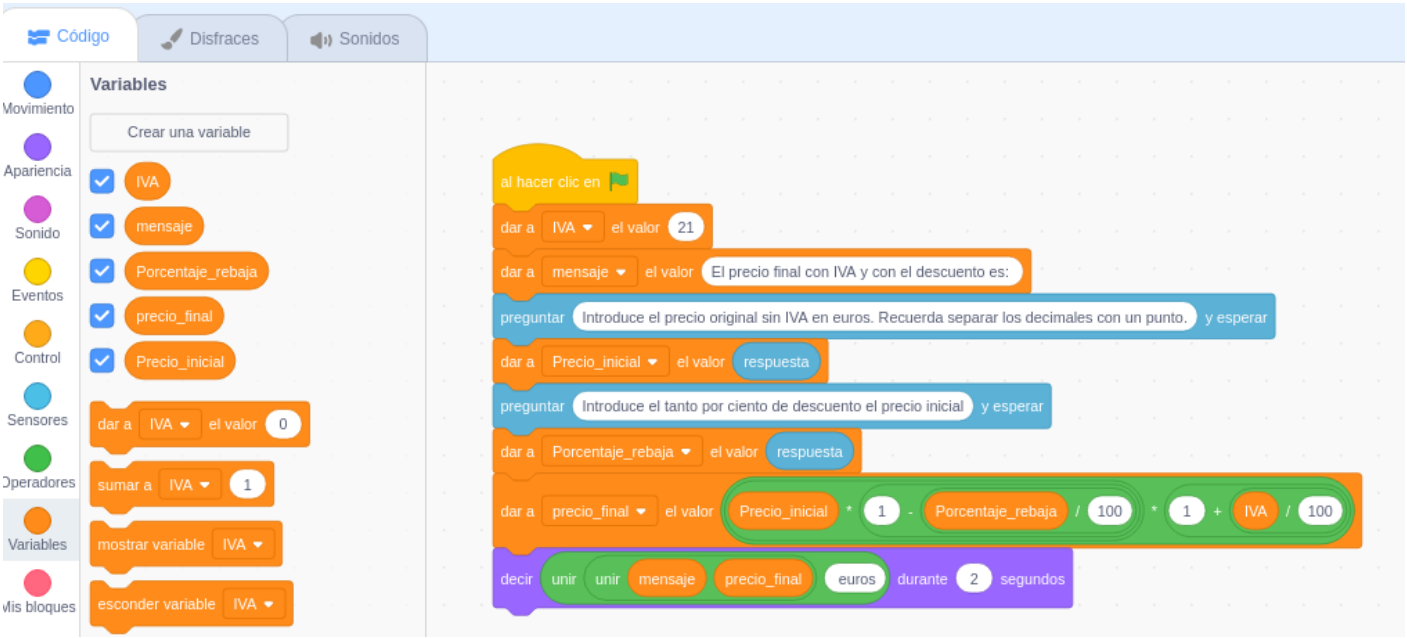
En el paso de depuración de errores y verificación del programa puedes probar a introducir errores posibles, para ver cómo reaccionaría el programa: por ejemplo meter los decimales con una coma, o un porcentaje de descuento decimal. En programación es fundamental adelantarse a equívocos o posibles confusiones de la persona usuaria, para tener prevista una respuesta que no bloquee el programa o prevenirlas mediante mensajes de aviso.

Pasos 3, 4 y 5: Codificación, compilación y verificación del programa *Calculadora de Rebajas* con Scratch

En Scratch, las estructuras de almacenamiento de datos se encuentran en el bloque **Variables**. Desde allí podremos crear tantas variables como necesitemos, y también asignarles el valor deseado mediante el bloque correspondiente.

Al crear una variable lo primero que nos solicita aparte de su identificador es saber si se define como **local** (solo de este objeto) o como **global** (común para todos los objetos) En este curso solo desarrollamos programas de un mismo objeto por lo que nos daría igual.

Una vez definidas las variables necesarias pasaremos a crear el código en el objeto utilizando **sensores**, **apariencia** y los **operadores** aritméticos y de concatenación que necesitemos, y que veremos con más detalle en el siguiente apartado. El código en bloques quedaría de la siguiente forma:



Pruébalo aquí:

https://scratch.mit.edu/projects/750474716/embed

Estructuras complejas de almacenamiento de datos

Aunque no es objeto de este curso, en programación se pueden almacenar **conjuntos de datos** siempre que sean **del mismo tipo** en otro tipo de estructuras complejas, los llamados **arrays** que en español se traducen como vectores o matrices, y en PSeInt se denominan **dimensiones**.

Para más información sobre cómo utilizar dimensiones en PSeInt se puede consultar el siguiente video.

https://www.youtube.com/embed/889nGzmU2yA

En Scratch solo se contempla la utilización de **listas** que serían series ordenadas de datos del mismo tipo, y que se encuentran disponibles dentro del mismo bloque **Variables**.

Financiado por el Ministerio de Educación y Formación Profesional y por la Unión Europea - NextGenerationEU

Financiado por la Unión Europea NextGenerationEU
Plan de Recuperación, Transformación y Resiliencia
GOBIERNO DE ARAGON

Procesamiento

Tipos de operaciones

Las operaciones a realizar con los datos pueden ser de muy diversa naturaleza:

- **Aritméticas**: operaciones clásicas de suma, resta, multiplicación y división.
- **Lógicas**: comparaciones, negación, Y, O.
- **Concatenación:** unión de varios elementos (cadenas de caracteres o variables de diferentes tipos)
- **Bucles**: Implica la realización de acciones de forma repetida. En este caso convendrá distinguir dos tipos:
  - Número de veces a repetir la acción conocido previamente: usaremos las estructuras **Para** (en inglés **For**) o **Repetir**.
  - Número de veces a repetir dependiendo de valores obtenidos: usaremos las estructuras **Mientras** (While...do) o **Repetir hasta que** (Do ...while), según deseemos evaluar la condición antes o después de la primera iteración. Estas estructuras veremos que son condicionales además de repetitivas.
- **Condicionales**: implica la realización de unas acciones u otras tomando decisiones. Estructuras **Si-entonces** (If-else), **Según (**Switch**)**.

Los tres primeros tipos de operaciones ya los hemos ido viendo en los apartados anteriores. Nos centraremos ahora en las dos últimas , si bien introduciremos diversas operaciones en los ejemplos para profundizar en su uso.

Iteraciones y bucles

Para practicar con estas estructuras, realizaremos un pequeño programa que nos calcule el promedio de varios números. En primer lugar el programa solicita la cantidad de números a promediar, luego pide que se introduzcan los números tantas veces como le hayamos dicho (aquí está la repetición) Al final muestra el promedio. En este caso, como el número de iteraciones es conocido, usaremos la instrucción **Para.** Asimismo, utilizaremos **operadores aritméticos** y de **concatenación**.

Pasos 1 y 2: Análisis y diagrama de flujo del programa *Promedio de n números*

Los elementos implicados serán:

- **Inicio y fin** de algoritmo.
- **Salidas**: Solicitar número de calificaciones, mostrar promedio.
- **Entradas**: cantidad de números y números a promediar.
- **Almacenamiento**: número de elementos (número entero), números introducidos (número real), acumulado de la suma de números (número real) y promedio (número real)
- **Procesamiento**: iteraciones, sumas y división.

El diagrama de flujo asociado a la instrucción **PARA** es el siguiente:

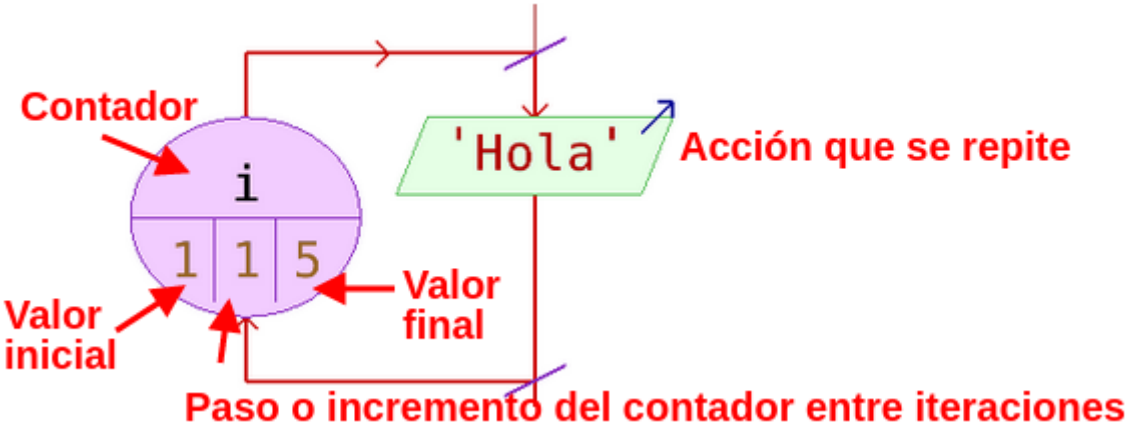
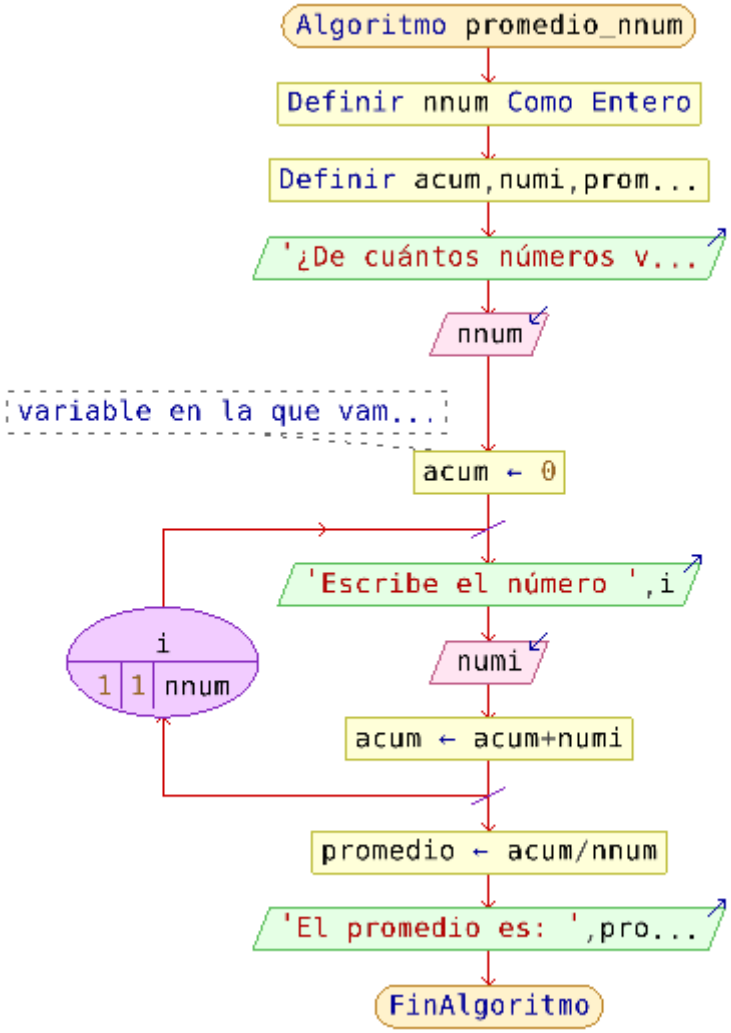


Diagrama de flujo:



Pasos 3, 4 y 5: Codificación, compilación y verificación del programa *Promedio de n números* con PSeInt

Como vimos en el apartado anterior, después de renombrar el algoritmo comenzamos por definir las variables implicadas escribiendo la expresión **Definir**

```
Algoritmo promedio_nnum
    Definir nnum Como Entero;
    Definir acum, numi, promedio Como Real;
```

PSeInt permite definir varias variables del mismo tipo en la misma línea, separadas con comas. Como veis, hemos definido una única variable *numi* donde guardaremos el número introducido por la persona usuaria cada vez.

A continuación solicitamos la cantidad de números a promediar (**Escribir**), leemos la respuesta y la almacenamos en la variable correspondiente (**leer**) e inicializamos el valor de acum en 0, aunque en realidad no sería necesario puesto que es el valor asignado por defecto.

```
Escribir "¿De cuántos números vas a calcular el promedio?";
leer nnum;
acum<-0; //variable en la que vamos a ir sumando todos los números
```

Con el valor de *nnum* definido por el usuario llega el momento de la repetición: solicitaremos al usuario que introduzca un número nnum veces. Para ello usaremos el comando **Para** disponible en la ventana de la derecha.

Al hacer clic sobre el comando, se nos escriben las siguientes instrucciones en nuestra hoja de algoritmo

```
Para variable_numerica<-valor_inicial Hasta valor_final Con Paso paso Hacer
    secuencia_de_acciones
Fin Para
```

Como *variable numérica* habitualmente se define una variable **local** (i, j, k...) que hace de contador. El *valor inicial* es 1 y el *valor final* el número de veces que queremos repetir la instrucción. En *paso* se especifica el crecimiento del contador de una iteración a otra (en nuestro caso de uno en uno)

Por último en la *secuencia de acciones* hemos de poner que es lo que queremos que se repita en cada ocasión. En nuestro caso solicitar el número, leerlo y sumarlo a los anteriores guardando el resultado en la variable *acum*.

```
Para i<-1 Hasta nnum Con Paso 1 Hacer
    Escribir "Escribe el número ", i;
    Leer numi;
    acum<-acum+numi;
Fin Para
```

El valor de *numi* se refresca y varía en cada iteración, puesto que ya hemos guardado el valor del número anterior en el acumulado.

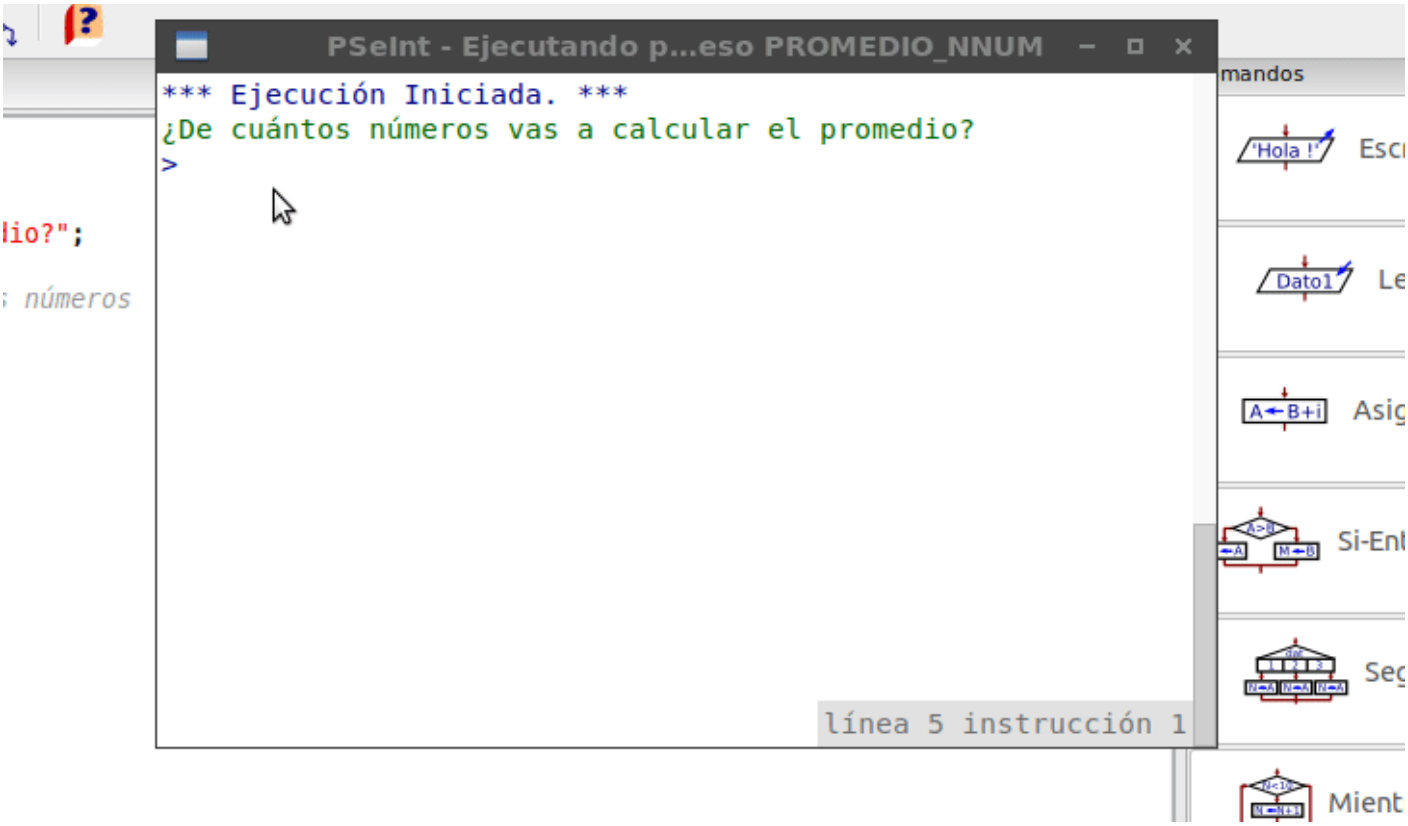
> En un programa es muy importante usar el **mínimo número de variables** precisas para no añadir complejidad innecesaria al mismo. (Principio KISS de diseño de software)

Hemos concatenado en el mensaje de solicitud la variable numérica de contador. Eso nos permite mostrarle al usuario en qué iteración se encuentra. La instrucción **Escribir** seguida de los elementos a concatenar separados por comas los coloca **en la misma línea y sin separación entre ellos**. Por eso es preciso considerar los espacios de separación en las cadenas de caracteres que incluyamos.

Por último, asignamos el valor del promedio al cálculo correspondiente y mostramos el resultado nuevamente concatenando dos mensajes.

```
    promedio<-acum/nnum;
    Escribir "El promedio es: ",promedio;
FinAlgoritmo
```

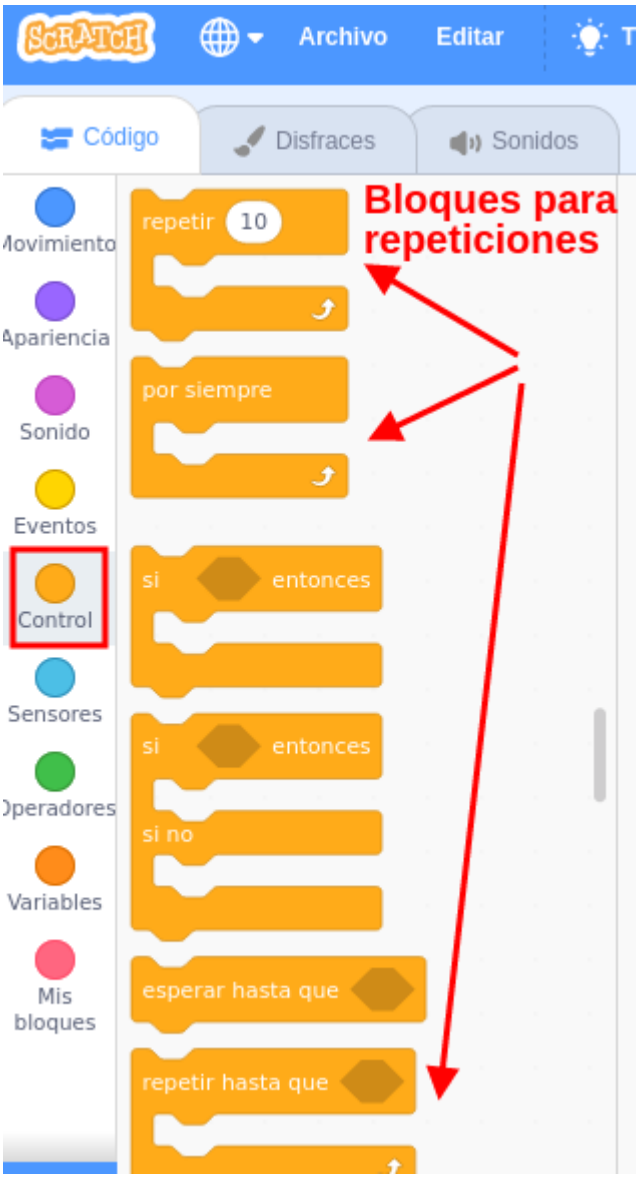
Finalmente le damos a **Ejecutar** y verificamos el proceso realizado.



Nuevamente en la verificación del programa podemos experimentar la robustez del programa frente a fallos, introduciendo números decimales con coma, negativos, etc...Invitamos a realizarlo e introducir las instrucciones necesarias para prevenir dichos fallos.

Pasos 3, 4 y 5: Codificación, compilación y verificación del programa *Promedio de n números* con Scratch

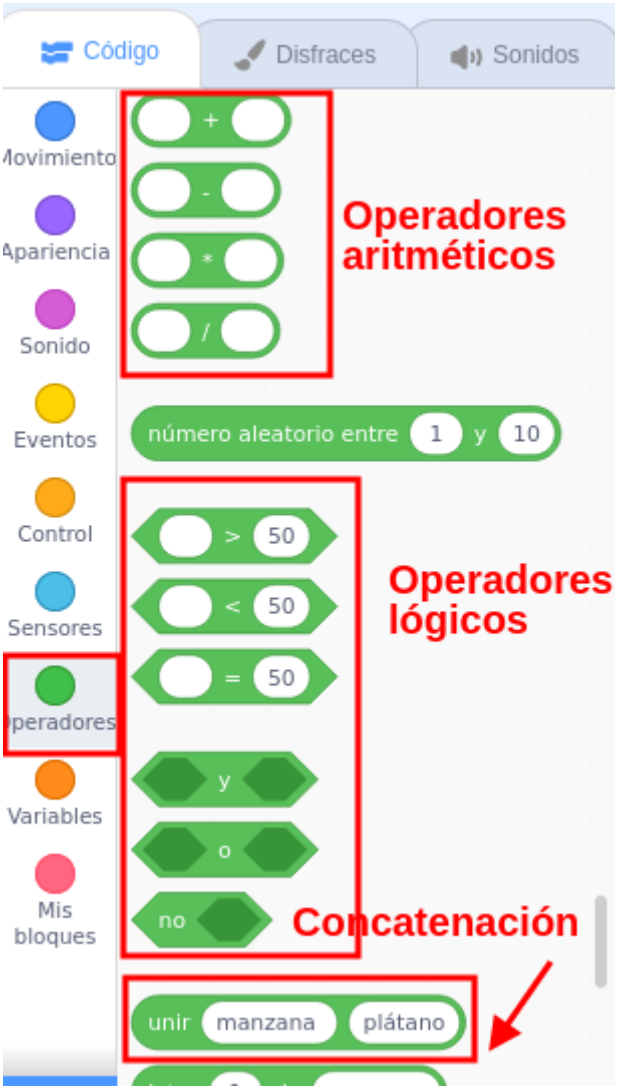
En Scratch los bloques relacionados con las estructuras repetitivas se encuentran en **Control** y son *Por siempre, Repetir y Repetir hasta que.*



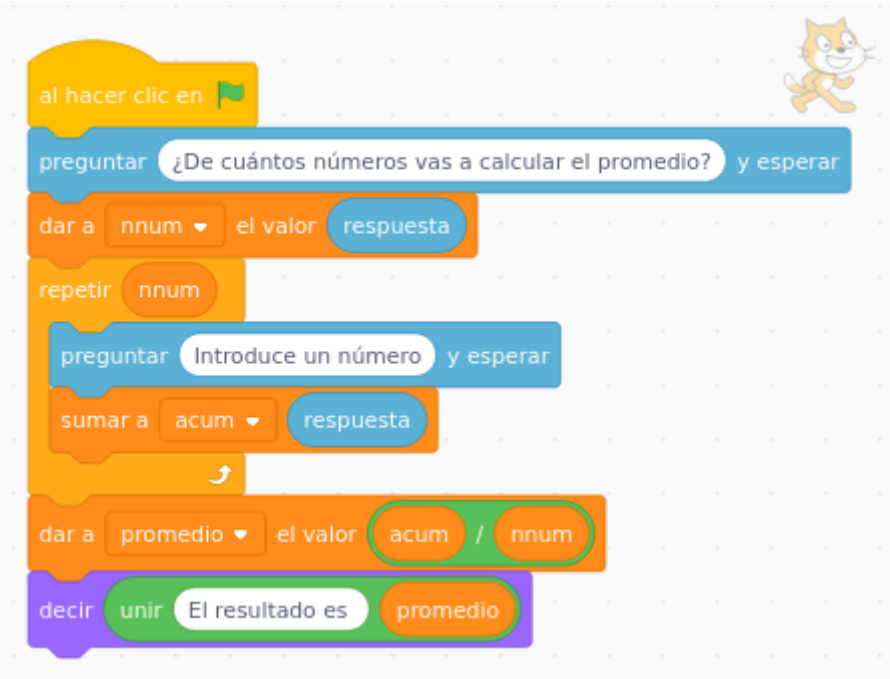
En nuestro caso, como el número estará definido, usaremos **Repetir.**

En primer lugar en el bloque **Variables** definiremos las variables necesarias.

De forma análoga, utilizando los bloques de **Sensores, Apariencia, Variables** explicados en los apartados de Entradas, Salidas y Datos, construimos el programa. En Operadores encontraremos los bloques necesarios para la realización de las operaciones aritmético, lógicas y de concatenación.



El programa finalmente con todos los elementos quedaría así:



Condicionales

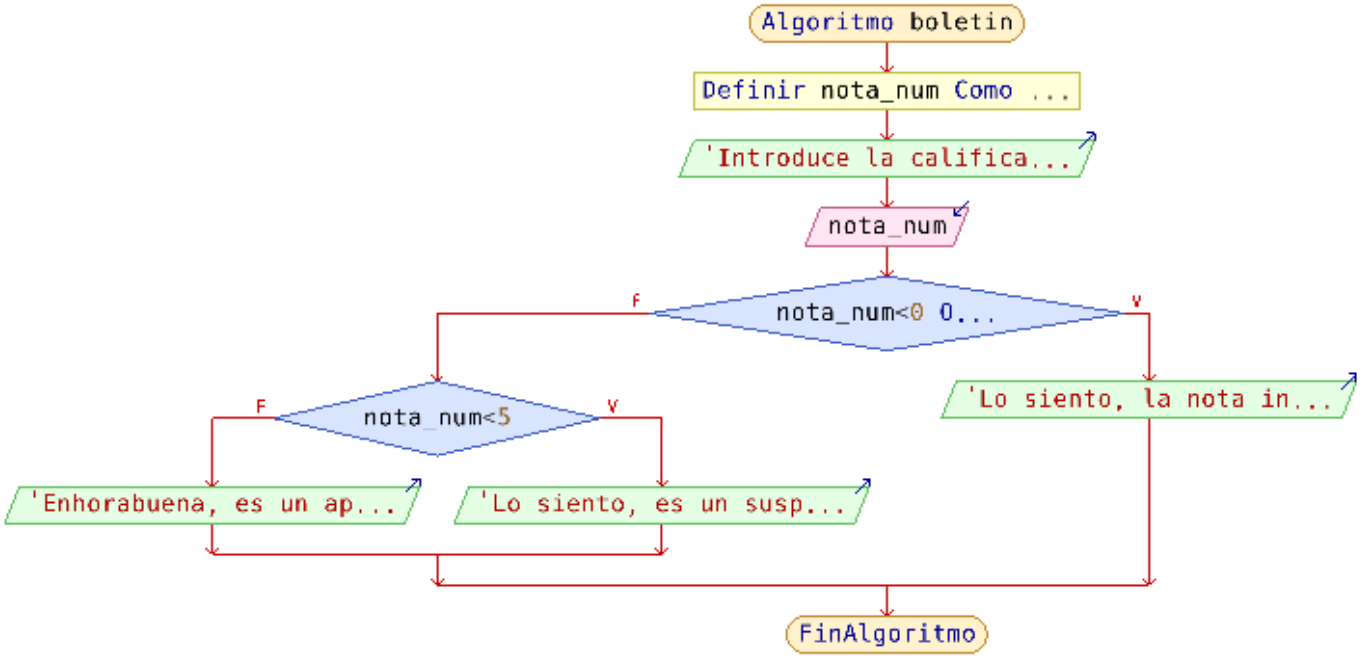
Para practicar con estas estructuras, realizaremos un pequeño programa en el que tras solicitar una nota numérica, y verificar en primer lugar si el valor recibido es correcto, nos indicará si la nota corresponde a un APROBADO o un SUSPENSO.

Pasos 1 y 2: Análisis y diagrama de flujo del programa *Boletín*

Los elementos implicados serán:

- **Inicio y fin** de algoritmo.
- **Salidas**: Solicitar calificación numérica, mostrar calificación textual o mensaje de error.
- **Entradas**: calificación numérica
- **Almacenamiento**: calificación numérica (número real, puede tener decimales)
- **Procesamiento**: lógicas (comparación,Y,O) y condicionales.

Diagrama de flujo:



En este caso hemos definido la condición de error mediante el operador lógico **O**, indicando que se considere errónea cualquier calificación menor que cero **o** mayor que 10. Este programa puede ser resuelto de forma análoga utilizando el operador **Y**, indicando que considere válida cualquier nota numérica mayor o igual que cero **y** menor o igual que 10.

Pasos 3, 4 y 5: Codificación, compilación y verificación del programa *Boletín* con PSeInt

En primer lugar renombramos el algoritmo (*boletin* en nuestro caso) y definimos las variables implicadas que solo es la calificación numérica (*nota_num*)

A continuación solicitamos la calificación numérica (Escribir) y la introducimos en la variable definida (Leer)

```
Algoritmo boletin
    Definir nota_num como Real;
    Escribir "Introduce la calificación numérica. Tiene que ser un número real entre 0 y 10. Si usas decimales recuerda poner un punto";
    Leer nota_num;
```

Ahora vendría la aplicación de la condición. Los comandos encargados en PSeInt de introducir las condiciones son **Si-entonces, Según, Mientras y Repetir**. Un ejemplo de la sintaxis de cada una se puede apreciar en la siguiente figura

```
Algoritmo Condicionales_PSeInt

    Si expresion_logica Entonces
        acciones_por_verdadero
    SiNo
        acciones_por_falso
    Fin Si

    Segun variable_numerica Hacer
        opcion_1:
            secuencia_de_acciones_1
        opcion_2:
            secuencia_de_acciones_2
        opcion_3:
            secuencia_de_acciones_3
        De Otro Modo:
            secuencia_de_acciones_dom
    Fin Segun

    Mientras expresion_logica Hacer
        secuencia_de_acciones
    Fin Mientras

    Repetir
        secuencia_de_acciones
    Hasta Que expresion_logica

FinAlgoritmo
```

- **Si-entonces**: verifica que se cumple una condición o no y en función de ello ejecuta unas acciones. Se pueden anidar unas condicionales dentro de otras.
- **Según**: se utiliza cuando una variable puede adoptar un número discreto de opciones (tipo menú) y se define el comportamiento ante ellas y ante el caso de que no coincida con ninguna.
- **Mientras**: evalúa una condición en bucle y mientras es verdadera ejecuta una acción. Mezcla condicional y bucle.
- **Repetir-hasta que** : ejecuta una acción en bucle hasta que una condición se cumple en cuyo momento se interrumpe. Mezcla condicional y bucle.

En nuestro ejemplo la condición **solo se evalúa una vez** por lo que descartamos las dos últimas, y el valor numérico introducido puede tener **infinitos valores** entre 0 y 10 por lo que la segunda tampoco es válida. Usaremos entonces el comando **Si-entonces.**

Vamos a aprovechar para introducir dos nuevos operadores, el operador **Y** representado por el carácter **&** y el operador **O** representado por la doble barra **||** que se utilizan cuando queremos que se evalúe más de una condición simultáneamente. En nuestro caso podemos considerar dos opciones:

- Si la nota introducida es menor que 0 **o** mayor que 10, no tiene sentido,y hay que dar un mensaje de error. En caso contrario habrá que distinguir (*condicional anidada*) si es menor que 5 y entonces el mensaje dicta suspenso, y si es mayor o igual que 5 que será aprobado.

```
Si nota_num<0 || nota_num>10 Entonces
    Escribir "Lo siento, la nota introducida no es válida";
SiNo
    Si nota_num<5  Entonces
        Escribir "Lo siento, es un suspenso.";
    SiNo
        Escribir "Enhorabuena, es un aprobado.";
    Fin Si
Fin Si
```

- Si la nota introducida es mayor o igual que 0 **y** menor o igual que 10, es correcta y hay que distinguir (*condicional anidada*) si es menor que 5 o mayor. En caso contrario sacar el mensaje de error.

```
Si nota_num>=0 & nota_num<=10 Entonces

    Si nota_num<5  Entonces
        Escribir "Lo siento, es un suspenso.";
    SiNo
        Escribir "Enhorabuena, es un aprobado.";
    Fin Si

SiNo
    Escribir "Lo siento, la nota introducida no es válida";
Fin Si
```

Las dos son correctas y elegir una u otra solo dependerá de la persona programadora y si prefiere usar operadores **Y** u **O**.

```
Algoritmo boletinOR
    Definir nota_num como Real;
    Escribir "Introduce la calificación numérica. Tiene que ser un número real entre 0 y 10. Si usas decimales recuerda poner un punto";
    Leer nota_num;


    Si nota_num<0 || nota_num>10 Entonces
        Escribir "Lo siento, la nota introducida no es válida";
    SiNo
        Si nota_num<5  Entonces
            Escribir "Lo siento, es un suspenso.";
        SiNo
            Escribir "Enhorabuena, es un aprobado.";
        Fin Si
    Fin Si
```

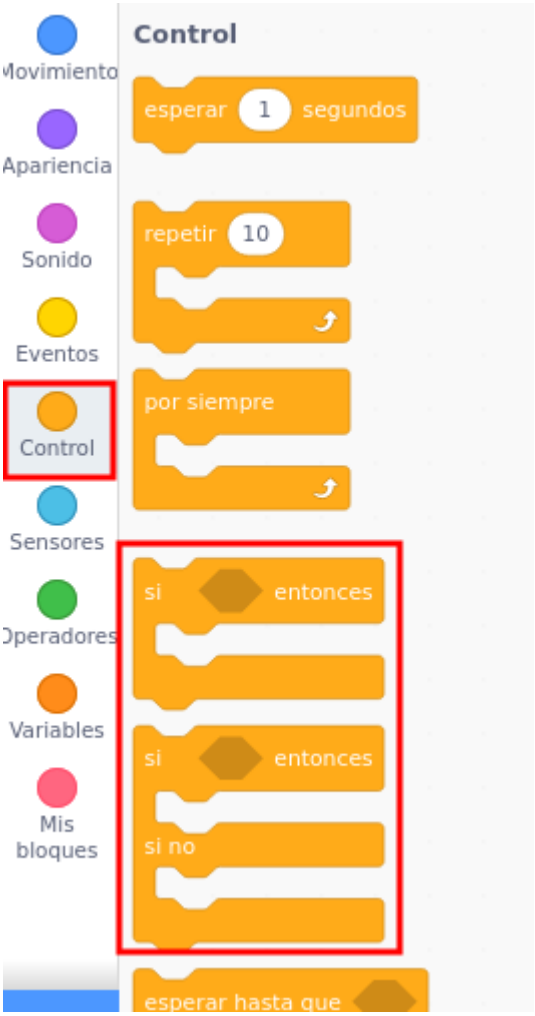
```
Algoritmo boletinAND
        Definir nota_num como Real;
        Escribir "Introduce la calificación numérica.Tiene que ser un número real entre 0 y 10. Si usas decimales recuerda poner un punto"
        Leer nota_num;


        Si nota_num>=0 & nota_num<=10 Entonces

            Si nota_num<5  Entonces
                Escribir "Lo siento, es un suspenso.";
            SiNo
                Escribir "Enhorabuena, es un aprobado.";
            Fin Si

        SiNo
            Escribir "Lo siento, la nota introducida no es válida";
        Fin Si

FinAlgoritmo
```

Finalmente sólo nos quedaría **Ejecutar** el programa y verificar su correcto funcionamiento.

Dejamos como ejercicio de ampliación el reformular este programa para que siga preguntando hasta que reciba una calificación válida. Como pista sugerimos explorar las posibilidades de la instrucción **Repetir**.

Pasos 3, 4 y 5: Codificación, compilación y verificación del programa *Boletín* con Scratch

En Scratch los bloques relacionados con las estructuras condicionales se encuentran en **Control,** junto a los bucles.



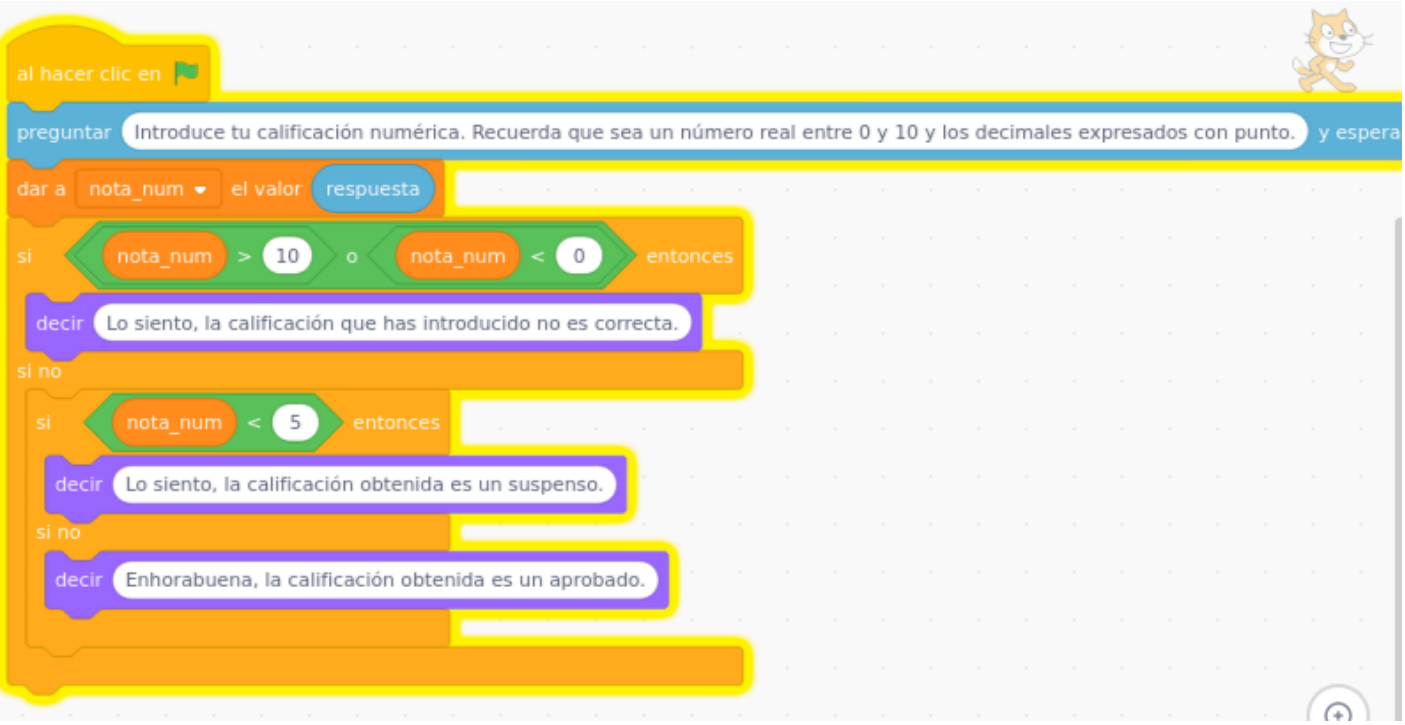
Los operadores lógicos Y y O se encuentran en **Operadores**, junto con los de comparación y los ya vistos anteriormente: aritméticos, de concatenación...

Como en todos los programas empezaremos definiendo las variables necesarias, en nuestro caso sólo una desde los bloques de **Variables**.

A continuación solicitaremos la calificación mediante el bloque disponible en **Sensores** y almacenaremos su respuesta en la variable recién creada.

Por último evaluaremos la respuesta obtenida mediante el bloque correspondiente de **Control** y utilizando los operadores lógicos de **Operadores**, según sea su valor ofreceremos el mensaje de error o escribiremos la calificación final con el bloque correspondiente de **Apariencia.**

El código resultante sería el siguiente:



En este caso hemos utilizado el operador **O** porque Scratch no dispone de los operadores combinados menor o igual y mayor o igual, y así nos ahorramos el tener que combinar ambos operadores con un operador lógico O. Recordad el principio KISS.

Por último, verificamos el correcto funcionamiento del programa y su robustez frente a los errores más comunes. En Scratch **ejecutamos** haciendo clic sobre la bandera verde.

Pruébalo aquí !!

https://scratch.mit.edu/projects/749701636/embed

Financiado por el Ministerio de Educación y Formación Profesional y por la Unión Europea - NextGenerationEU

Funciones

A veces en nuestros programas existen acciones que hemos de realizar varias veces a lo largo de los mismos, y que nos resultaría farragoso tener que copiar el código una y otra vez. De igual forma, cualquier modificación en algún elemento tendría que buscarse y repetirse todas las veces en que ese código saliera. Para automatizar todo esto cumpliendo el principio DRY de programación nos ayudan las **funciones.**

Las **funciones,** también llamados **métodos** o **procedimientos**, empaquetan y 'aíslan' del resto del programa una parte de código que realiza alguna tarea específica de forma que nos sea muy sencillo de manipular y reutilizar. Se definen al principio como pequeños subprogramas y después se "invocan" desde el algoritmo principal cuando se necesitan.

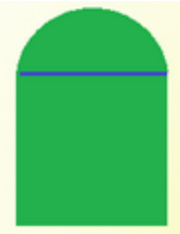
Las funciones pueden pueden devolver o no un resultado y también ser con o sin parámetros.

- Funciones que **devuelven resultado o no**:
  - Si la función no devuelve resultado en ocasiones se denomina **procedimiento** y es un conjunto de código con algo de relación que se invoca en el algoritmo principal de forma autónoma. Por ejemplo, la función *dibujarCuadrado (lado)* puede realizar todas las operaciones necesarias para dibujar un cuadrado dado su lado. En el algoritmo principal escribiríamos.
    - dibujarCuadrado(5);
  - Si la función devuelve resultado se comportará como un valor y siempre habrá que **almacenar dicho resultado en una variable**, por lo que se habrá de invocar dentro de una asignación y no de forma autónoma. Por ejemplo, la función *calcularAreaCuadrado(lado)* calcula la superficie de un cuadrado dada la longitud de su lado, pero en el programa principal deberá aparecer asignándose a una variable u ocupando el lugar de ella.
    - Resultado=calcularAreaCuadrado(5);
- **Parámetros** de una función: valores que necesita la función para poder ejecutarse. Por ejemplo una función que dibuja un cuadrado necesita que le pasen el valor del lado para dibujarlo. Sin embargo una función que realice una cuenta atrás del 10 al 1 no necesita ningún parámetro.En ese caso se escribe el paréntesis pero vacío.
  - dibujarCuadrado(10);
  - contarAtras10a1();

> Se podría parametrizar la función contarAtras (n) indicándole a la función el número a partir del que realizar la cuenta atrás. Te invitamos a explorar cómo sería.

Las funciones se suelen denominar con un verbo o un conjunto de varias palabras que comienza con un verbo en minúsculas; es decir, con la primera letra en minúscula y la primera letra de las palabras siguientes en mayúsculas.

Por ejemplo, vamos a imaginar un programa que calcula el área de una figura geométrica plana compuesta como la de la figura.

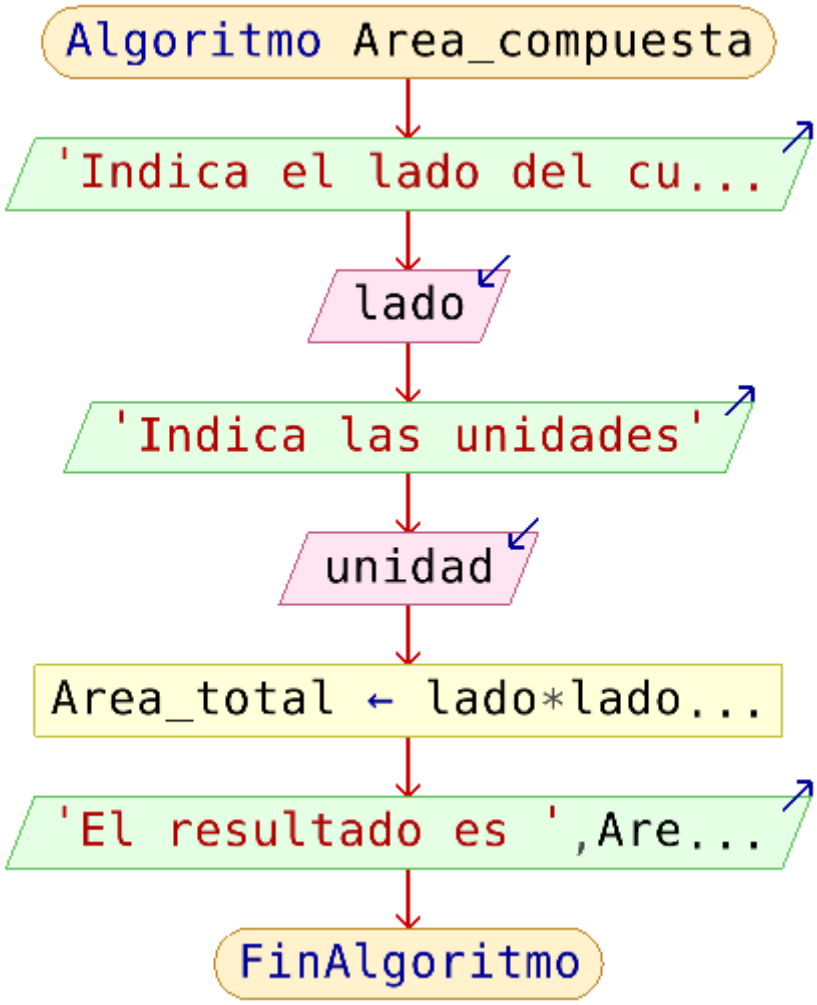


Pasos 1 y 2: Análisis y diagramas de flujo del programa *Area compuesta*

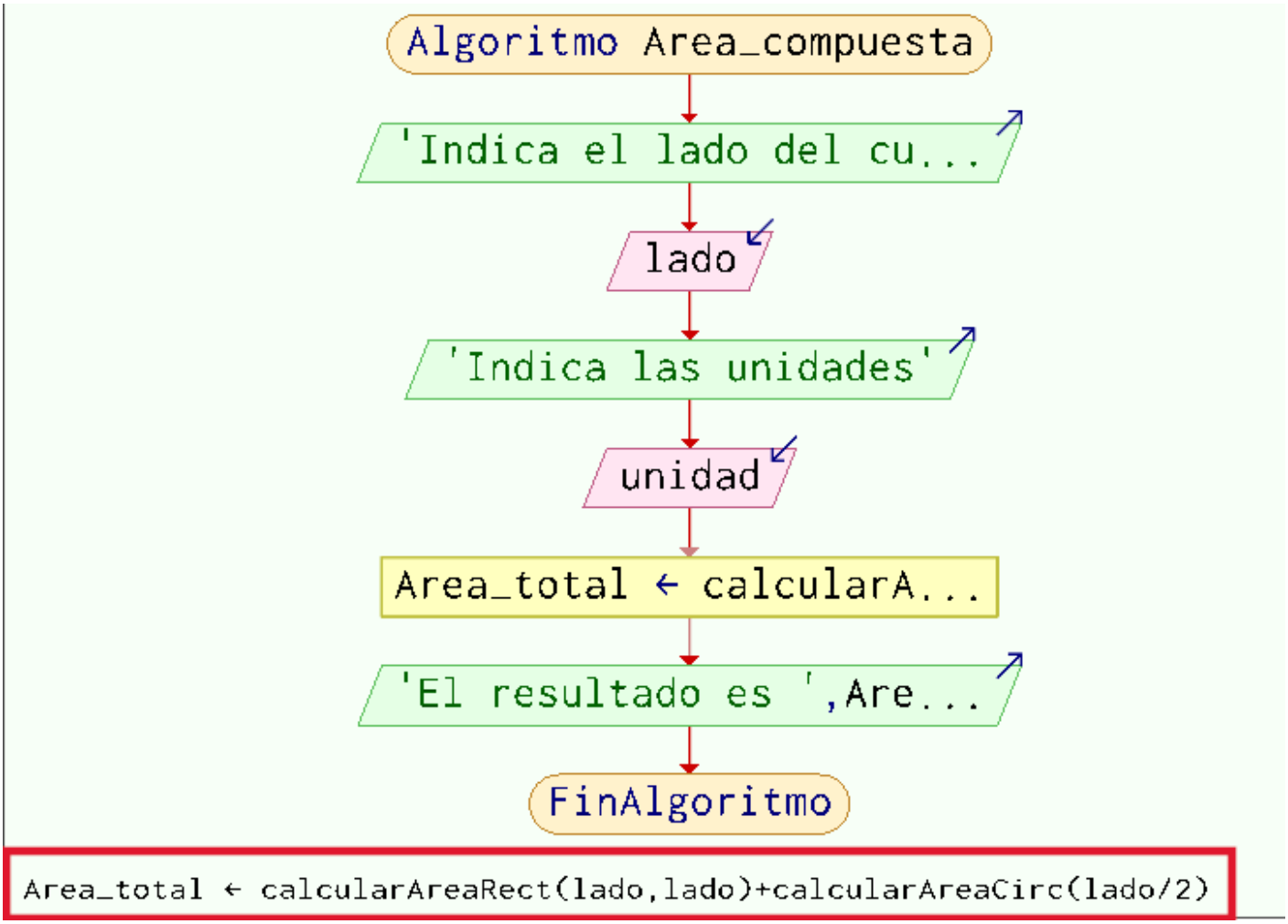
Los elementos serían:

- **Inicio y fin** del algoritmo.
- **Salida**: pedir las medidas del lado del cuadrado inferior incluyendo unidades.
- **Entrada**: longitud del lado (que coincide con diámetro del círculo) y unidad de medida.
- **Almacenamiento**: lado, unidad, area_total.
- **Procesamiento**: suma, multiplicación, concatenación.

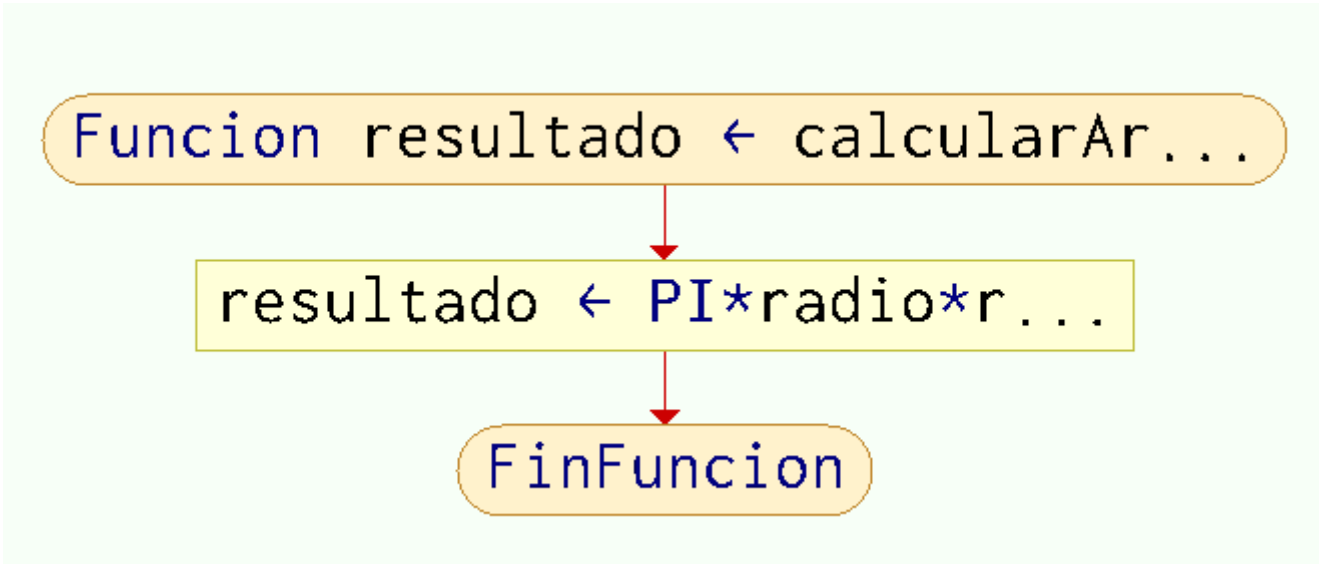
Diagrama de flujo (sin funciones):

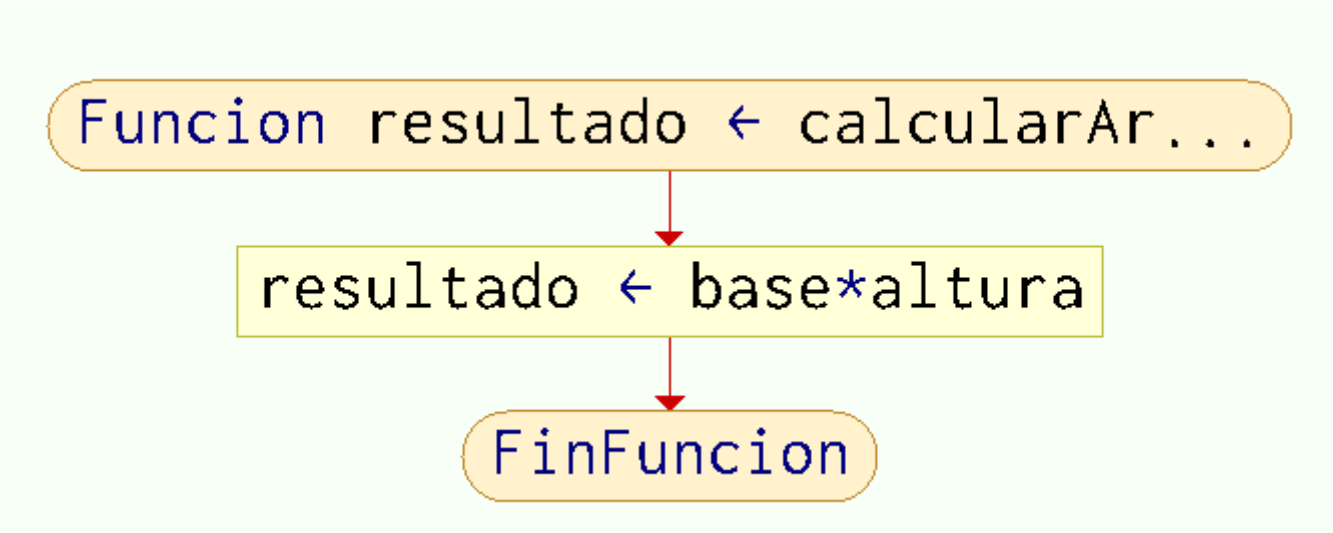


Otra forma de abordar su solución sería predefinir unas funciones que nos calcularan por ejemplo el área de un rectángulo y el área de un círculo. Veamos como quedaría el diagrama de flujo en ese caso.



Y habría que realizar de forma independiente los diagramas de flujos de cada una de las funciones.

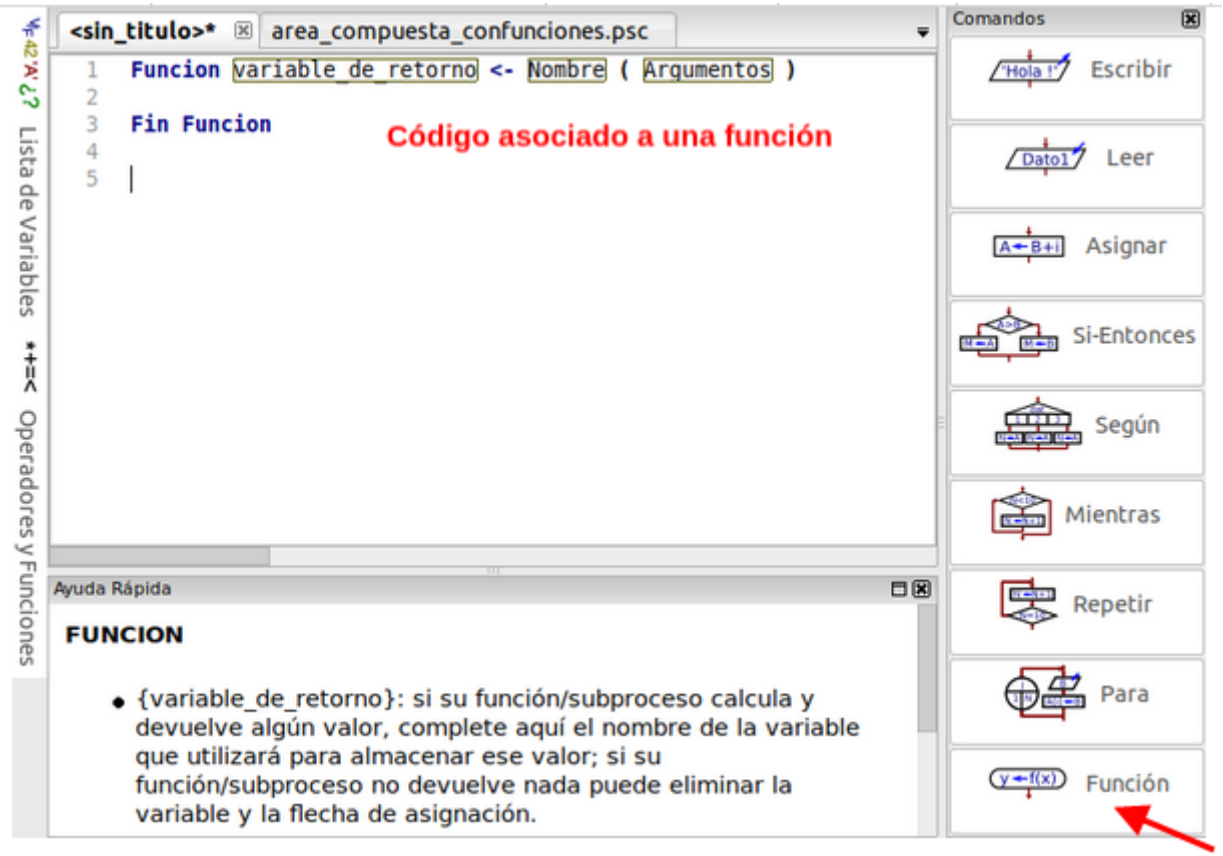




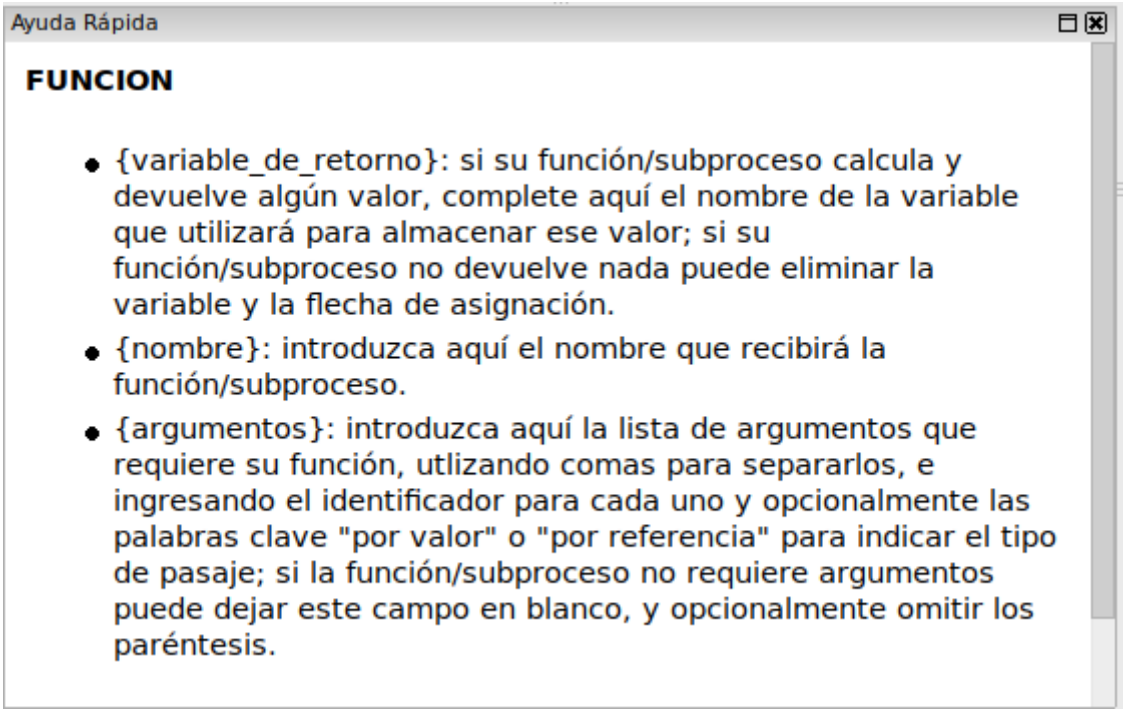
Las funciones necesarias en este caso devuelven un resultado, y tienen parámetros. Como se puede deducir fácilmente, esto tendría mucho más sentido en programas donde fuera necesario calcular áreas de diferentes figuras de forma repetida.

Pasos 3, 4 y 5: Codificación, compilación y verificación del programa *Area_compuesta* con PSeInt.

En PSeInt para introducir funciones se definen al principio del código y utilizando el comando **Función**.



En la ventana de ayuda rápida nos explica cada uno de los elementos que aparecen en el código.



- **Variable de retorno**: sólo tiene sentido si la función devuelve un valor. Si no, se borra.
- **Nombre**: con el que posteriormente se invocará a la función en el algoritmo principal.

- **Argumentos**: o parámetros necesarios para la función.

Cambiamos pues el nombre del fichero y definimos dos funciones al inicio, una para calcular el área de un rectángulo y otra para calcular el área de un círculo. Ambas devuelven resultado y necesitan parámetros, la primera la base y la altura, y la segunda el radio. En PSeInt existe de forma predefinida la constante PI con el valor *3.1415926536* por lo que podemos usarla y no habrá que definirla.

area_compuesta_confunciones.psc

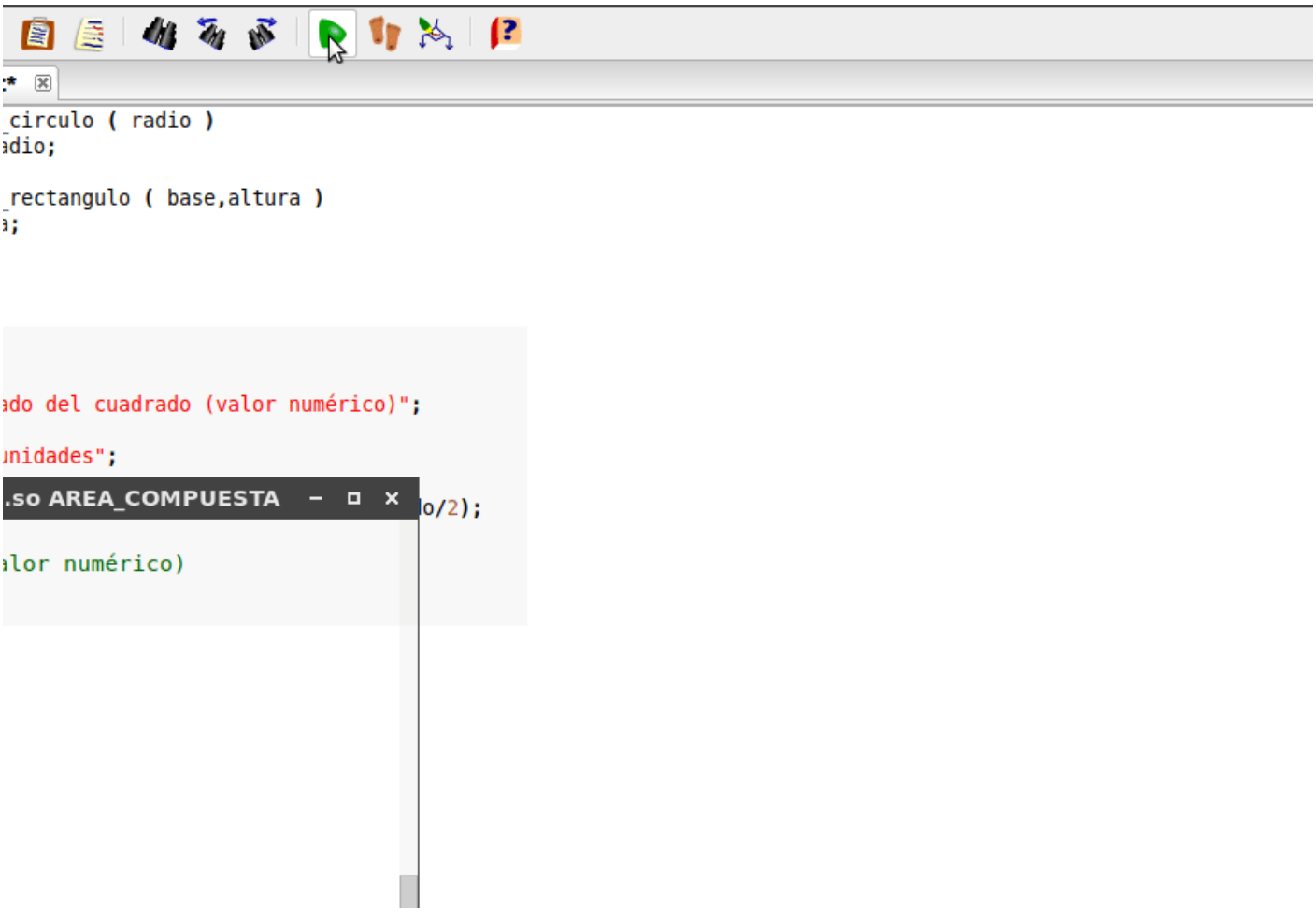
```
1  Funcion resultado ← calcularAreaCirc ( radio )
2      resultado←PI*radio*radio;
3  Fin Funcion
4  Funcion resultado ← calcularAreaRect( base,altura )
5      resultado←base*altura;
6  Fin Funcion
7
```

Una vez definidas las funciones, pasaremos a escribir el algoritmo principal. El argumento de las funciones será el valor introducido por el usuario y que corresponde con el lado del cuadrado.

```
Algoritmo Area_compuesta

    Escribir "Indica el lado del cuadrado (valor numérico)";
    Leer lado;
    Escribir "Indica las unidades";
    Leer unidad;
    areaTotal←calcularAreaRect (lado,lado)+calcularAreaCirc(lado/2);
    Escribir "El resultado es ",areaTotal,unidad,"2";

FinAlgoritmo
```

Al ser funciones que devuelven un resultado hemos tenido que definir una variable areaTotal que almacene el resultado de lo obtenido por las funciones. De forma más correcta y generalizable tendríamos que haber predefinido previamente las variables lado y areaTotal como números reales, y unidad como string, si bien PSeInt nos permite ser un poco más laxos en el procedimiento. La última línea es la concatenación del valor obtenido, la unidad introducida y un 2 puesto que la unidad será elevada al cuadrado.

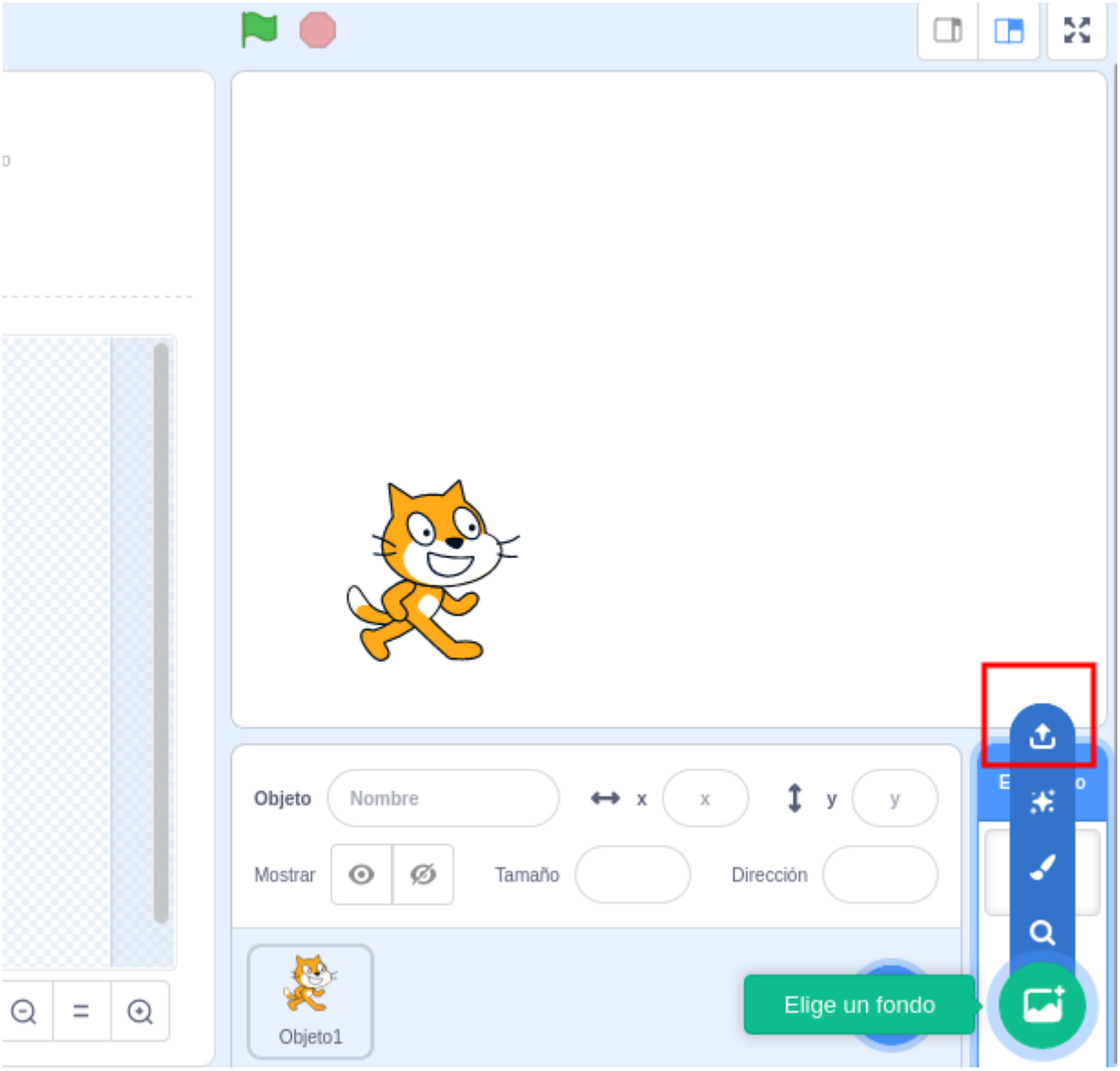
Solo nos queda ejecutar el programa y verificar su funcionamiento.



Pasos 3, 4 y 5: Codificación, compilación y verificación del programa *Area_compuesta* con Scratch.

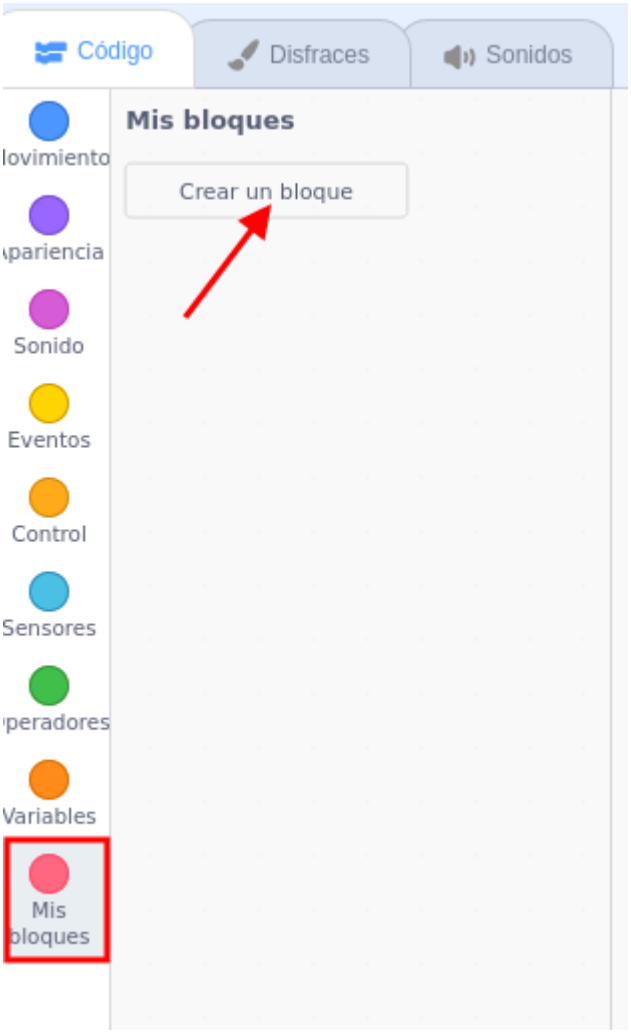
En Scratch no existe de forma explícita la opción de crear funciones que devuelven un resultado. Solo existe lo que hemos llamado procedimientos, eso sí con o sin parámetros. Eso se hace mediante la creación de nuevos bloques. Para crear funciones que devuelvan un resultado tendremos que crear dichos bloques y asignarles dentro del mismo el valor calculado a una variable predefinida. Veámoslo con el ejemplo.

En primer lugar cargaremos la imagen de la figura cuyo área queremos calcular como fondo del escenario. Para ello en la parte inferior derecha hacemos clic sobre el botón verde y escogemos la opción de cargar un escenario.



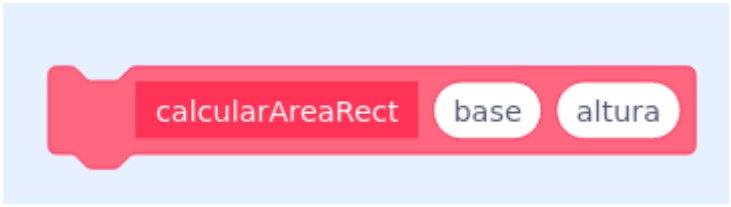
A continuación crearíamos las variables necesarias. Como en este programa las únicas variables necesarias son el lado del cuadrado y su unidad, y eso hemos visto que va a ser almacenado por defecto como respuesta del usuario, no es necesario. Tampoco vamos a necesitar crear una variable areaTotal para almacenar el resultado puesto que lo mostraremos directamente concatenando operadores y así nos ahorraremos un bloque.

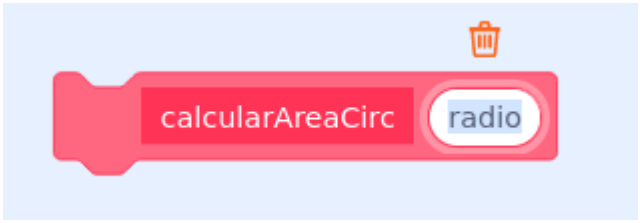
A continuación crearemos las funciones calcularAreaRect(base, altura) y calcularAreaCirc (radio) Para ello Scratch nos permite crear nuevos bloques a los que les asociemos un código. Eso se hace desde **Mis Bloques** y pinchando en **Crear un bloque.**



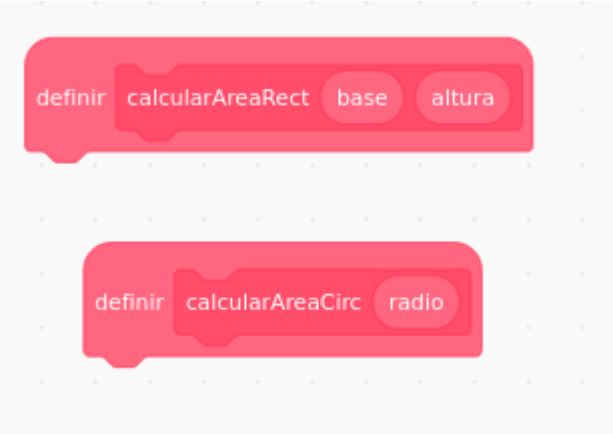
Al clicar sobre Crear un bloque se nos abre una ventana donde elegir el identificador de la función/bloque, así como opciones de añadirle argumentos o parámetros a esa función. Pueden ser texto, número, variable booleana o simplemente una etiqueta de texto.

En nuestro caso crearemos dos nuevos bloques. El del área del rectángulo requiere dos entradas numéricas (base y altura), y el del área del círculo una única entrada numérica (radio).



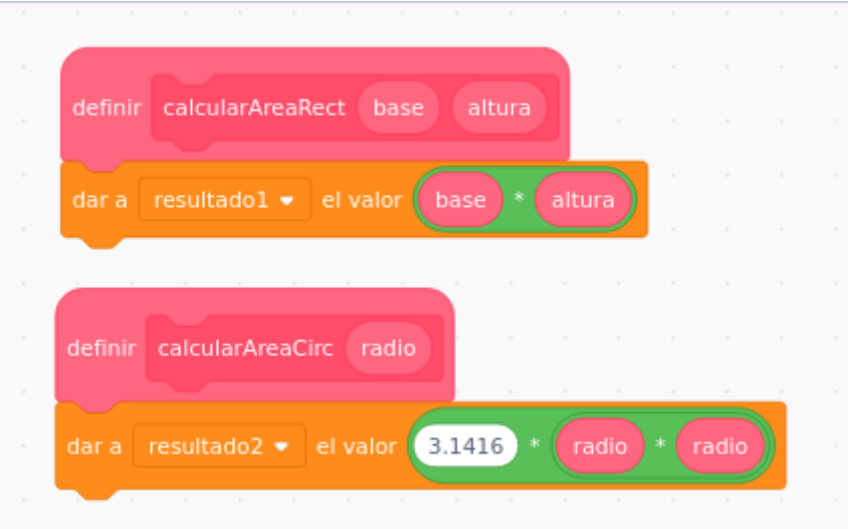


En ese momento en nuestra área de código aparecen dos nuevos elementos para que definamos bajo ellos los bloques que van a componer cada uno de estos subprocedimientos.



Como hemos dicho antes, son funciones que devuelven un resultado, por lo que primero habremos de crear variables para poder almacenar en ellas ese resultado. Les llamaremos resultRect y resultCirc.

Mediante operadores aritméticos en este caso, añadiremos los bloques correspondientes al cálculo del área de cada figura, dados sus parámetros.



Por último, una vez definidas las funciones/bloques, solo nos quedará el algoritmo principal, que tendrá este aspecto:

La última línea incluye la concatenación de texto con los resultados parciales de las funciones creadas. Solo quedaría comprobar su funcionamiento y depurar posibles errores.

Prúebalo aquí.

https://scratch.mit.edu/projects/750180283/embed

Financiado por el Ministerio de Educación y Formación Profesional y por la Unión Europea - NextGenerationEU

Financiado por la Unión Europea NextGenerationEU
Plan de Recuperación, Transformación y Resiliencia
GOBIERNO DE ARAGON

Permitidme un comentario...

A lo largo de los ejercicios y ejemplos anteriores habrás visto que en algunos casos han aparecido comentarios en los programas.

Un **comentario en programación** es un texto en nuestro código fuente que el compilador ignora. Y si lo ignora, ¿por qué lo ponemos? Habitualmente para darnos explicaciones internas de cuál es la función de cada parte del programa. También son muy útiles para descartar fragmentos de código de forma temporal, sin tener que borrarlos, y así facilitar su posterior recuperación.

Los comentarios en programación son fundamentales:

- Facilitan la comprensión: no solo de otros usuarios sino del mismo programador/a a lo largo del tiempo que dura el desarrollo del programa.
- Favorecen la colaboración: vuelven el código comprensible para otras personas coprogramadoras que de esa forma pueden participar también.
- Facilitan la depuración de errores: comentar y descomentar fragmentos de código nos permitirá detectar en qué lugar se están produciendo los fallos.

En muchas ocasiones bastará como ejercicio que a nuestro alumnado le propongamos un programa y le digamos que lo organice y le coloque comentarios. Eso nos dará idea de si entiende realmente el funcionamiento del mismo, paso previo a que pueda más adelante adaptar ese programa a nuevas necesidades e incluso crear uno desde cero. Pero esto lo veremos con más detalle en el módulo siguiente.

Comentarios en PSeInt

Para realizar comentarios en PSeInt simplemente tendremos que encabezar el punto en el que queremos insertar el comentario con una doble barra. Eso nos pondrá el texto que viene a continuación en un gris pálido y en cursiva que nos indicará que el compilador no va a tomar en cuenta lo escrito de esta forma.

```
Fin Funcion
Funcion resultado ← calcularAreaRect( base,altura )
    resultado←base*altura;
Fin Funcion

//Defino el algoritmo principal

Algoritmo Area_compuesta

    Escribir "Indica el lado del cuadrado (valor numérico)";
    Leer lado;
    Escribir "Indica las unidades";
    Leer unidad;
    areaTotal←calcularAreaRect (lado,lado)+calcularAreaCirc(lado/2);
    Escribir "El resultado es ",areaTotal,unidad,"2";

FinAlgoritmo
```

También se pueden introducir los comentarios al final de una línea de código.

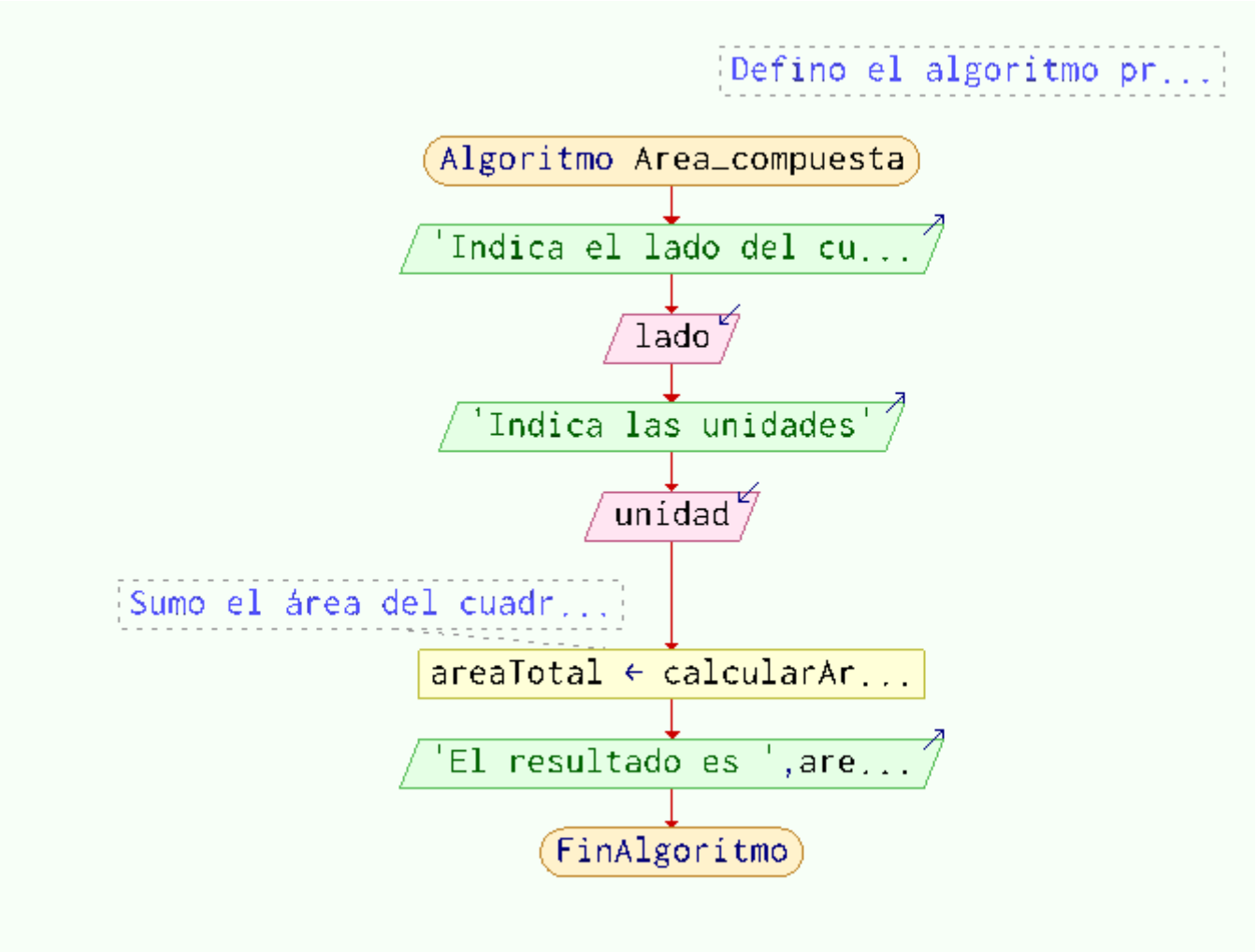
```
Leer unidad;
areaTotal←calcularAreaRect (lado,lado)+calcularAreaCirc(lado/2); //Sumo el área del cuadrado y del semicírculo
Escribir "El resultado es ",areaTotal,unidad,"2";
```

PSeInt solo permite comentar líneas y no fragmentos enteros de código, por lo que si queremos descartar un conjunto de instrucciones deberemos comentar línea por línea todas las que constituyan el bloque. Eso obviamente no sucede en lenguajes estructurados de programación que sí poseen esta capacidad.

```
Funcion resultado ← calcularAreaCirc ( radio )
    resultado←PI*radio*radio;
Fin Funcion

//Funcion resultado ← calcularAreaRect( base,altura )
    //resultado←base*altura;
//Fin Funcion
```

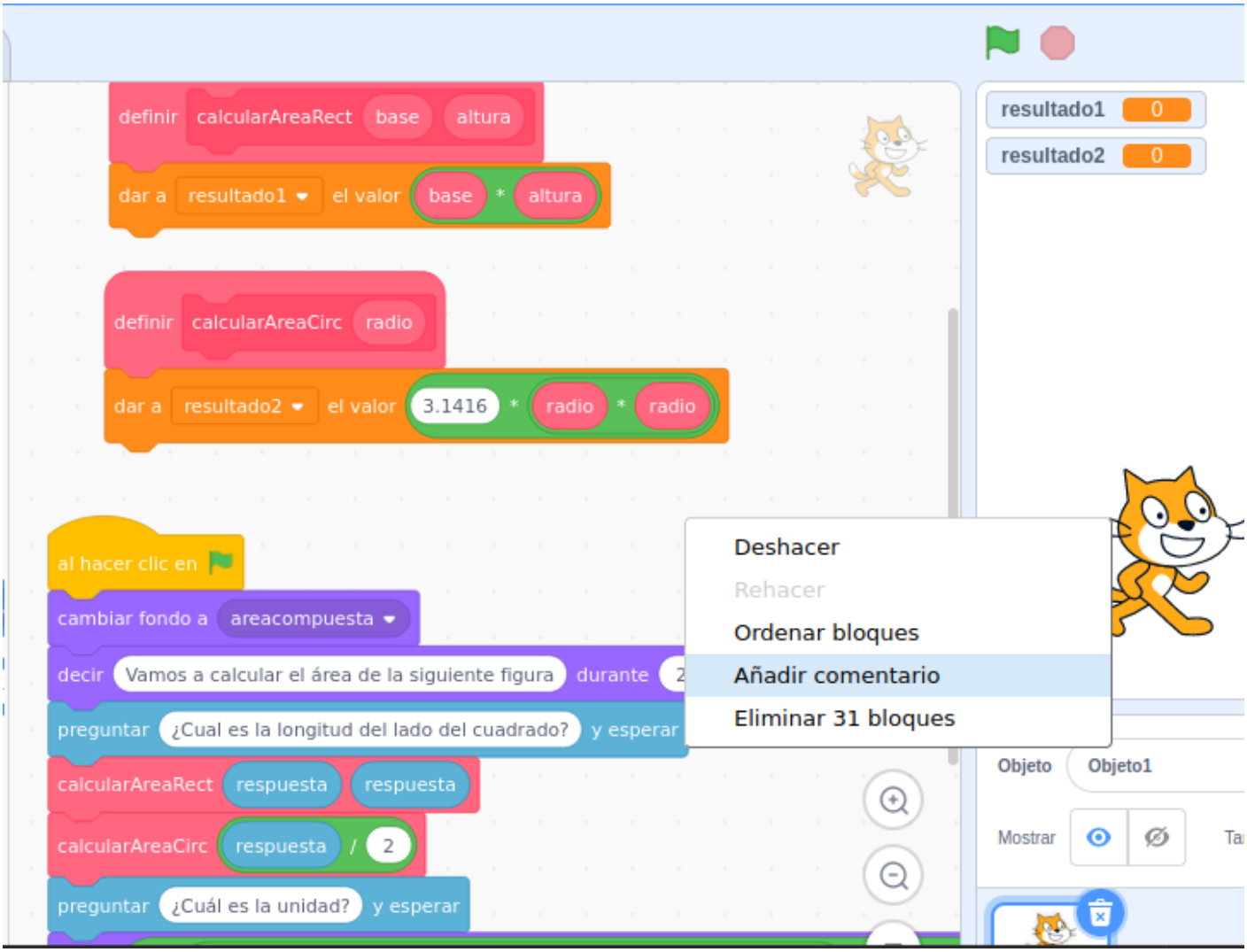
Esta capacidad es especialmente útil en las fases de depuración de fallos, para aislar los fragmentos de código donde se encuentran los errores.

Los comentarios por supuesto no aparecen en la ventana de ejecución del programa. Dónde sí son visibles es en el diagrama de flujo asociado.

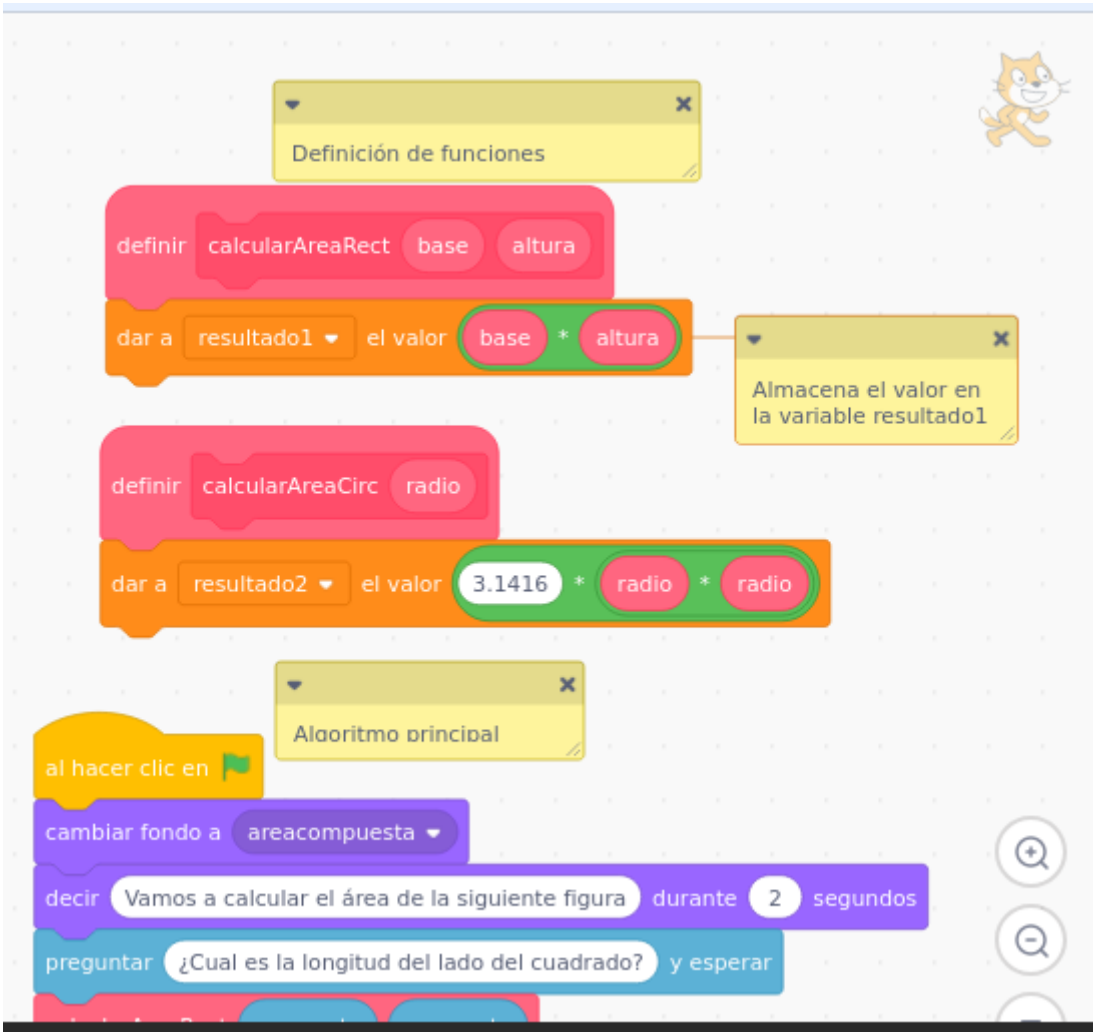


Comentarios en Scratch

En Scratch se pueden añadir comentarios desde la ventana de edición de código asociado a cada objeto. Los comentarios pueden estar tanto asociados a un bloque como sobre la misma ventana de edición. Se añaden clicando con el botón derecho sobre el lugar donde queremos insertar el comentario.



Una vez añadidos, quedan como notas visibles en el área de edición del código.



Por supuesto en Scratch no existe la opción de comentar y descomentar bloques para que no sean tenidos en cuenta por el compilador. La opción disponible para realizar esto es separar el evento que determina la ejecución de ese bloque y el código quedará temporalmente desactivado.

Financiado por el Ministerio de Educación y Formación Profesional y por la Unión Europea - NextGenerationEU