TEMA 2

Conceptos básicos de programación

PARTE I. Resolución de problemas y especificación de algoritmos

1. Resolución de problemas.

En este tema presentamos ciertas nociones elementales que intervienen en la tarea de diseñar algoritmos. En el paradigma de programación que utilizaremos, el imperativo, se entiende un proceso como una sucesión de cambios de estado en el entorno en el que estamos trabajando. Se define un **procesador** como una entidad capaz de ejecutar unas acciones dentro del marco de este entorno particular. Se distinguen las **acciones primitivas**, que el procesador es capaz de realizar inmediatamente, de otras acciones que deben descomponerse en acciones primitivas. También se muestra cómo especificar algoritmos describiendo las condiciones iniciales y finales que dicho algoritmo debe satisfacer.

2. Procesador, Entorno y Acciones.

Procesador:

Elemento o entidad capaz de entender un enunciado y de ejecutar el trabajo indicado.

Entorno:

Conjunto de objetos necesarios para la ejecución del trabajo. El entorno, para un procesador dado, es específico del trabajo a realizar. Trabajo y entorno están estrechamente ligados. En función del entorno podrá realizarse el trabajo o no.

Generalmente, la ejecución de un trabajo supone la realización de una serie de etapas para llegar al fin deseado.

Acción:

Suceso que modifica el entorno. En este caso, cada una de las etapas para la realización del trabajo.

Acción primitiva:

Para un procesador dado, si el enunciado de dicha acción es suficiente para que el procesador pueda entenderla y ejecutarla sin información complementaria.

Una acción no primitiva debe descomponerse en acciones primitivas.

Estado:

Situación del entorno en un determinado instante.

Resultado:

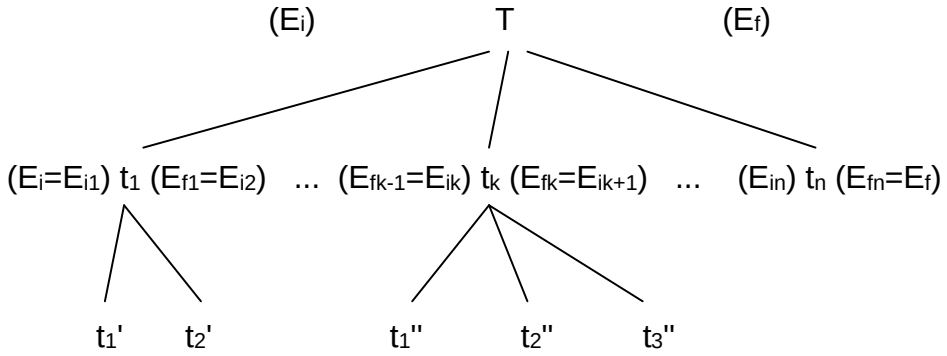
Estado del entorno en el momento final.

3. Análisis descendente.

El análisis descendente es un método para descomponer una acción no primitiva en un conjunto de acciones primitivas.

Dado un enunciado T descrito por un enunciado no primitivo, el análisis descendente de T cosiste en encontrar una descomposición t_1, t_2, ..., t_n que sea una sucesión de enunciados cuya ejecución realice el trabajo T.

Como vemos en la figura siguiente, cada trabajo t_k hace cambiar el entorno de un estado E_{ik} a un estado E_{fk}.



Para cada enunciado t_k, hay dos casos posibles:

- t_k es una acción primitiva y se detiene el análisis descendente para t_k
- t_k no es una acción primitiva y se descompone en un nuevo t_k

En la medida en que cada trabajo sólo depende del trabajo precedente por el estado que espera, puede decirse que los trabajos de la descomposición son relativamente independientes y, en la mayoría de los casos, se pueden descomponer independientemente unos de otros.

Algoritmo:

Dado un procesador bien definido y un problema a resolver por este procesador, un algoritmo de resolución del problema es el enunciado de una secuencia de acciones primitivas que dejan el entorno en un estado final solución del problema.

Ejemplo.

Tenemos que realizar la agrupación de un documento de 10 páginas que ha sido enviado a un servicio de fotocopiado.

Para cada una de las páginas del documento, el servicio de fotocopiado ha hecho alrededor de 100 ejemplares. La incertidumbre en el número es debida a que la fotocopiadora no funciona correctamente.

Disponemos de una pila de 10 paquetes de hojas fotocopiadas, donde cada paquete está formado por 100 hojas aproximadamente de la misma página. Los paquetes no están dispuestos en la pila en por orden de página. Se quieren agrupar las fotocopias por un individuo (procesador) que dispone de los siguientes objetos (entorno) para efectuar su tarea:

- 10 paquetes de hojas numeradas
- un escritorio donde están apiladas, paquete a paquete, las hojas que se han de ordenar
- una grapadora
- 10 mesas numeradas del 1 al 10 . La mesa número i recibirá las hojas con el número i.

Finalmente, supondremos que el individuo encargado del trabajo solamente es capaz de realizar las siguientes acciones (primitivas):

- coger y poner
- ir de una mesa a otra (pasar de la mesa i a la i+1)
- ir a una mesa de número concreto (se supone que el individuo es capaz de reconocer la igualdad entre dos números: el de la mesa y un número dado)
- reconocer la presencia o ausencia de hojas sobre una mesa
- grapar

El trabajo a realizar puede ser descrito por la frase siguiente:

A partir de la pila que hay en el escritorio, agrupar las fotocopias.

Esta acción no es, evidentemente, una acción primitiva para nuestro individuo. Una solución por refinamientos sucesivos podría llevarnos a una primera descomposición tal como esta:

A1: Repartir los paquetes a razón de un paquete por mesa, paquete i en la mesa i.

A2: Reunir 10 hojas, graparlas y ponerlas en el escritorio.

Las acciones A1 y A2 no son acciones primitivas, por tanto habrá de hacerse una nueva descomposición en acciones más elementales:

Repetir

A11: ir al escritorio

A12: coger el paquete situado encima de la pila sobre el escritorio

A13: ir a la mesa cuyo número es el mismo que el de las hojas del paquete

A14: poner el paquete sobre la mesa

Mientras que A15: queden paquetes en el escritorio

Repetir

A21: coger las hojas necesarias para un ejemplar

A22: grapar el ejemplar

A23: ir al escritorio

A24: poner el ejemplar sobre el escritorio

Mientras que A25: queden hojas en todas las mesas

La acción A21 es una acción compuesta (no es primitiva). Seguimos con la descomposición (refinamiento) de acciones:

Repetir

A11: ir al escritorio

A12: coger el paquete situado encima de la pila sobre el escritorio

A13: ir a la mesa cuyo número es el mismo que el de las hojas del paquete

A14: poner el paquete sobre la mesa

Mientras que A15: queden paquetes en el escritorio

Repetir

A211: ir a la mesa número 1

A212: coger una hoja de la mesa

Repetir

A213: ir a la mesa siguiente

A214: coger una hoja de la mesa

Mientras que A214: la mesa no sea la nº 10

A22: grapar el ejemplar

A23: ir al escritorio

A24: poner el ejemplar sobre el escritorio

Mientras que A25: queden hojas en todas las mesas

La secuencia de acciones así obtenida utiliza sucesivamente dos esquemas "repetir ... hasta que" y además, el segundo esquema repetitivo contiene otro esquema "repetir ... hasta que" (esquemas anidados). El enunciado del problema ha sido puesto en forma de una secuencia de acciones primitivas. Hemos conseguido por tanto el algoritmo de resolución de ese problema.

4. Especificación de problemas

A nuestro nivel, la especificación de un problema va a consistir en un “enunciado preciso” del mismo. Se trata de describir el problema sin aludir a detalles relacionados con su resolución (el **qué** frente al **cómo**). Así, una especificación, dependiendo del grado de formalización que se plantee, se puede hacer:

a) En lenguaje natural.

b) Semiformal.

c) Formalmente mediante predicados o asertos.

En nuestra notación, la especificación de un problema constará de los siguientes dos elementos:

1) **Interfaz:**
   - Declaración de variables de entrada (datos)
   - Declaración de variables de salida (resultados).
2) **Efecto:**
   - Condiciones exigibles a los datos de entrada
   - Efectos producidos: relaciones entre los datos y los resultados y/o cambios producidos en el entorno.

En la declaración de las variables de entrada y salida indicaremos número de variables y sus tipos. Usaremos lenguaje natural en nuestras especificaciones, procurando evitar enunciados ambiguos. Una especificación nunca puede tener vacío el apartado **efectos producidos** (también conocido como postcondición).

Ejemplos

1- Especificar un algoritmo que obtenga la raíz cuadrada positiva de un valor real positivo dado.

Interfaz:

- Entrada: real x
- Salida: real r

Efecto:

- Condiciones previas: x >= 0
- Efecto producido: r^2= x, r >= 0

2- Especificar un algoritmo que determine si un entero positivo es potencia de 2

Interfaz:

- Entrada: entero n
- Salida: booleano esPotencia

Efecto:

- Condiciones previas: n >= 0
- Efecto producido:
  - esPotencia = verdad si n es potencia de 2
  - esPotencia = falso en caso contrario

Ejercicios

Dar especificaciones correctas para los siguientes problemas:

1) Convertir una cantidad entera positiva de segundos en su equivalente en horas, minutos y segundos.
2) Determinar el máximo de dos enteros.
3) Indicar si un entero es un cuadrado perfecto.
4) Indicar si un entero es múltiplo de otro dado.
5) Calcular la parte entera de la raíz cuadrada de un número entero.
6) Calcular el producto de dos enteros positivos.

5. Problemas, algoritmos y programas

Los problemas que van a centrar la atención de esta parte del curso son problemas de tratamiento de información. Muchos tipos de problemas de tratamiento de información constituyen campos de aplicación de la informática: cálculo o cómputo, gestión, control (de procesos industriales, de robots...), tratamiento de señales, biomédicos, lúdicos, de inteligencia artificial (reconocimiento de imágenes, reconocimiento de voz, lenguaje natural, planificación inteligente, sistemas expertos, etc.), etc.

El ordenador, tal como se vio en la parte anterior del curso, es la herramienta utilizada para resolver de forma automática problemas de tratamiento de información. El comportamiento de un ordenador viene parametrizado por un programa, cuya ejecución permite alcanzar la solución del problema considerado.

A partir de un problema de tratamiento de información ¿cómo obtener el programa que lo resuelva?. Para ello se propone proceder en dos fases.

Una primera fase de **análisis** que da como resultado una **solución del problema o algoritmo.** Un algoritmo consta **de una descripción de la información** asociada al problema y una **descripción del modo de tratamiento de esta información**. Un algoritmo debe ser expresado en un lenguaje que, siendo cómodo para el analista, no presente ambigüedades. En este curso utilizaremos un castellano "restringido" (seudocódigo) como notación algorítmica.

A partir de un algoritmo se debe abordar una segunda fase: **codificación.** Consiste en transcribir el algoritmo mediante un **lenguaje de programación** que, siendo adecuado al tipo de problema considerado, pueda ser procesado (interpretado o compilado) en el sistema informático del que se dispone. El resultado de esta fase es un **programa** (escrito en un determinado lenguaje) que consta de una **descripción de datos** y de una **descripción de acciones** a ejecutar. Las acciones se describen mediante composición de **instrucciones** ejecutables.

Más formalmente diremos que un **algoritmo** es una secuencia ordenada y finita de pasos, exenta de ambigüedades que lleva a la solución de un problema dado.

Propiedades de un algoritmo, no estrictamente necesarias, pero sí convenientes son:

- **Generalidad**, no debe restringirse a resolver problemas muy específicos
- **Eficiencia,** es decir, su consumo de recursos (especialmente tiempo de ejecución y memoria) debe ser "razonable".
- **Independencia** de la máquina y del lenguaje. El algoritmo debe poder ser programado de diferentes lenguajes y ser ejecutados en diversos ordenadores.

Un **programa** es un conjunto de instrucciones directamente comprensibles por un sistema de tratamiento de información que permite ordenar la ejecución de la secuencia de pasos elegida para resolver un problema dado.

La tarea de expresar un algoritmo en términos de un lenguaje "comprensible" por el ordenador se denomina **codificación.**

6. Elementos de un programa

Símbolos:

Palabras clave (if, while, main, do, etc.). Identificadores (de diversos objetos informáticos). Operadores (+, -, *, =, >, etc.). Separadores (, ; etc.)

Un identificador se construirá, en nuestra notación algorítmica, de acuerdo con la siguiente regla:

identificador ::= <letra> { <letra> / <dígito> / '_' }

Esta notación será utilizada a lo largo del curso. Los símbolos "/" permiten definir opciones alternativas, mientras que lo encerrado entre llaves puede reproducirse cero o más veces. La regla anterior dice que un símbolo identificador se construye como una secuencia de símbolos tal que el primero es una letra y los sucesivos, si existen, pueden ser letras o dígitos o el carácter '_'.

Sintaxis:

Indica cómo encadenar símbolos para formar "frases" (instrucciones) correctas.

Instr.Condicional ::= **if** <condición> **then** <acción1>
[**else** <acción2>]

En la regla anterior se utilizan los corchetes. Lo que éstos encierran puede aparecer cero o una vez en la frase descrita. Por lo tanto la regla sintáctica anterior dice que una *Instrucción Condicional* se construye necesariamente como una secuencia en la que se concatenan el símbolo ***if*** seguido de una *condición,* seguida del símbolo ***then*** y de un *bloque.* Esta secuencia puede, eventualmente, ir seguida por un símbolo ***else*** y otro *bloque.*

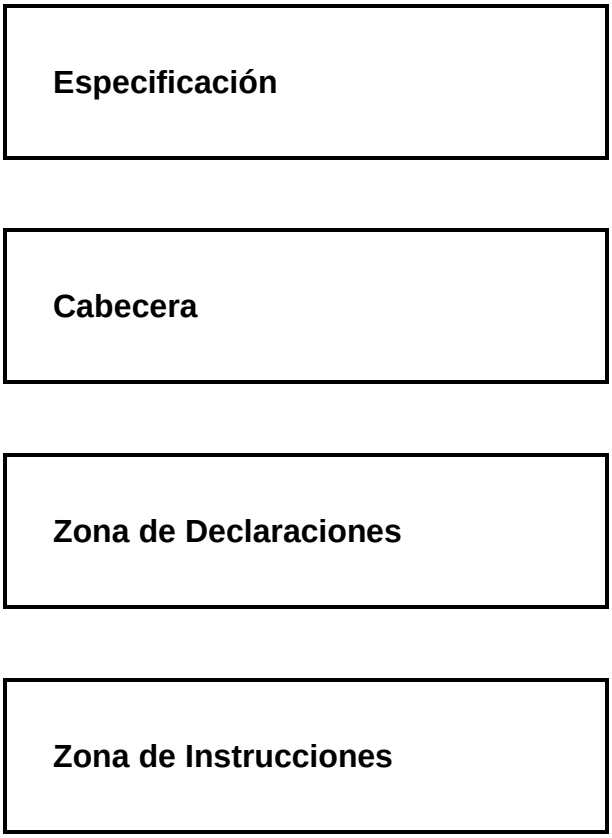
Semántica:

Expresa el significado (para el ordenador) de cada "frase". La semántica de la instrucción condicional anterior indica que se debe evaluar, en primer lugar, <condición>. Si el resultado es "verdad" entonces se ejecutará <acción1>, mientras que en el caso contrario se ejecutará <acción2>. Si en la instrucción ha sido omitida la cláusula *else* y <condición> es evaluada a "falso" no se ejecutará ninguna acción adicional.

7. Estructura de un algoritmo.

Hemos adelantado en el punto anterior que en el presente curso utilizaremos una notación en castellano "restringido" para escribir algoritmos. Es decir, no usaremos un lenguaje de programación concreto, sino lo que podríamos denominar un "seudocódigo". Significa esto que las palabras "clave" serán en castellano, aunque no por eso dejaremos de seguir unas reglas para la escritura de algoritmos.

Un algoritmo estará formado por los siguientes bloques:



El orden será siempre este y su contenido será el siguiente:

```
Especificación
Interfaz:    Entrada
             Salida
Efecto:      Condiciones previas
             Efecto producido
```

```
algoritmo  nombre
```

```
constantes
     lista de constantes
tipos
     declaración de tipos de datos
variables
     lista de variables
```

```
principio

     instrucciones (acciones)

  fin
```

PARTE II. Tipos de datos

1. Concepto de tipo de dato.

Un algoritmo trata información en forma de datos. Los datos representan una **abstracción** de la realidad. Un dato tiene asociado un valor. Con los datos se suele **operar.** Un dato debe ser **representado internamente** en un sistema informático. Conviene clasificar los datos por su "tipo".

Un tipo de dato viene caracterizado por:

- **Dominio**: Un conjunto de posibles valores**.**
- **Representación**: Un modo común de representar sus valores (tanto en los algoritmos como internamente en la memoria).
- **Operadores**: Conjunto de operadores asociados.

Trabajar con datos, clasificados por tipos, aumenta la fiabilidad de los programas y facilita la detección de algunos errores de programación, fundamentalmente errores de incompatibilidad entre operadores y operandos. Sean, por ejemplo, **letra** y **número** dos datos de tipo carácter y entero, respectivamente. La expresión **letra + número** es incorrecta y como tal podrá ser detectada por el procesador del lenguaje.

2. Tipos predefinidos

Señalamos ciertos tipos de datos que, dada su importancia, vamos a considerar que son proporcionados por cualquier entorno de programación, es decir, están ya definidos y presentan un comportamiento fijo. En particular, nos referimos a los tipos: entero, real, booleano, carácter y cadena (de caracteres).

Antes de describirlos uno a uno, es importante conocer que todos ellos van a disponer de una relación de orden total entre sus valores. Dos datos, A y B, de un mismo tipo, pueden ser comparados por los operadores binarios de relación:

==	Igual que	**!=**	Distinto de
<	Menor que	**<=**	Menor o igual que
>	Mayor que	**>=**	Mayor o igual que

El resultado de toda operación de relación es un valor de tipo booleano, verdad o falso.

2.1 Tipo Entero.

Dominio:

Su dominio está definido como el intervalo de valores enteros comprendidos entre dos valores constantes característicos de cada implementación concreta. Más concretamente, del espacio reservado para almacenar internamente un valor entero (2 bytes, 4 bytes, etc.).

Representación externa:

Los datos enteros se escribirán en un algoritmo (representación externa) en notación arábiga, eventualmente precedidos por un signo '+' o '-'. El signo se puede omitir si el valor es positivo.

456	-3156	+456	0
-18	22222	-2345	345

Representación interna:

Se suelen representar internamente en código binario complemento a 2, utilizando 2, 4 ó más bytes. En el caso de utilizar 2 bytes (16 bits), se tendrá:

0 = 0000 0000 0000 0000 1 = 0000 0000 0000 0001

-1= 1111 1111 1111 1111 -7= 1111 1111 1111 1001

32767 = 0111 1111 1111 1111 (maxentero = 2^{15} -1)

-32768 = 1000 0000 0000 0000 (minentero = - 2^{15})

Operadores:

- Las operaciones binarias de relación, con el significado habitual entre números enteros.
- Operaciones aritméticas binarias:

+	suma
-	resta

*	multiplicación
div	división entera (por defecto)
mod	resto de la división entera
/	división (da como resultado un real)

2.2 Tipo Carácter

Dominio:

Los caracteres alfabéticos: ‘A’, ‘B’, ..., ‘Z’, ‘a’, ‘b’, ...,’z’

Los caracteres numéricos o dígitos: ‘0’, ‘1’, ..., ‘9’

Otros caracteres imprimibles: ‘+’, ‘-’, ‘ ‘ , ‘;’ , etc.

Representación externa:

Externamente se escriben entre apóstrofos.

Representación interna:

Para su representación interna se suelen utilizar códigos numéricos. El código ASCII es el más usual. Un dato de tipo carácter se almacena internamente ocupando un byte (ocho bits).

Operadores:

Las operaciones binarias de relación (la definición de estos operadores relacionales viene dada por el código ASCII de los datos de tipo carácter). Por ejemplo, ‘3’<’4’ es verdad, mientras que ‘h’>’t’ es falso.

- La función ordinal **(ascii**) aplicada a un carácter, que da como resultado el valor entero que corresponde a su código ASCII. Entonces:

  ascii('A')=65 ascii('B')=66 ascii('0')=48
  ascii('1')=49 ascii(' ')=32 ascii('+')=43

- La función **chr**, inversa de la función ordinal, que, aplicada al código numérico asociado a un carácter da como resultado el propio carácter:

  chr(65)='A' chr(66)='B' chr(48)='0'

chr(49)='1' chr(32)=' ' chr(43)='+'

No obstante, el interés de las funciones **ascii** y **chr** es su utilización combinada o relativa, obviando los detalles de la codificación interna utilizada:

chr(ascii('B')+4)='F'

ascii(chr(i))=i

ascii('7')-ascii('0')=7

2.3 Tipo Booleano (tipo Lógico)

Dominio: Dos posibles valores, verdad o falso

Representación externa:

El identificador de los dos posibles valores: verdad o falso.

Representación interna:

Los datos lógicos se pueden codificar internamente con uno solo bit (falso=0, verdad=1). No obstante, en muchas implementaciones, por razones de acceso a memoria, se utiliza una mayor zona de almacenamiento (un byte o una palabra).

Operadores:

- Operaciones lógicas binarias:

  OR o-lógico o unión lógica

  AND y-lógico o intersección lógica

- Operaciones lógicas unitarias:

  NOT negación lógica

Conviene recordar que, tal como se avanzó previamente, las operaciones de relación proporcionan como resultado valores de tipo booleano.

2.4 Tipo Real

Dominio

Pretende representar al conjunto de los números reales. El dominio depende de la forma de representación interna, estando comprendido ente un valor mínimo y un valor máximo. Por supuesto, no se dispone de todos los números reales

entre dichos extremos, sino de un subconjunto finito de los mismos (nos puede servir la idea trabajar con un cierto grado de aproximación).

Representación externa

Escribiremos los valores de datos de tipo real en notación arábiga, precedidos por un signo '+' o '-'. Si se omite el signo se entiende que el número es positivo, o en notación científica:

+3.2 532.99 -12.43 0.0 45.3E4 23.21e10

Representación interna

Se suelen representar en forma **mantisa B**. La base B suele ser 2 o 4, 8 y 16. Según el número de bits que se reserven para almacenar el exponente podremos representar números más grandes o pequeños y según el número que se reserven para la mantisa podremos representar más o menos números reales (debido a la precisión de cifras decimales).

Operadores

- Las operaciones de relación.
- Operaciones aritméticas binarias
  - \+ suma
  - \- resta
  - * multiplicación
  - / división real
- Funciones aritméticas de librería. Supondremos definidas estas:

`abs(x)`	valor absoluto	`abs(-9.2)=9.2`
`sqrt(x)`	raíz cuadrada	`sqrt(28.09)=5.3`
`sin(x)`	seno (x en radianes)	`sin(3.9)=-0.6878`
`cos(x)`	coseno (x en radianes)	`cos(3.9)=-0.7259`
`arctang(x)`	arco tangente (resultado en radianes)	`arctang(3.9)=1.3198`
`ln(x)`	logaritmo neperiano	`ln(3.9)=1.3610`
`exp(x)`	exponencial e	`exp(3.9)=49.4025`

2.5 Tipo Cadena

Debido a su utilidad, lo vamos a considerar como un tipo de datos predefinido. Sin embargo, aunque todos los entornos de programación nos lo van a proporcionar, puede haber diferencias significativas de un sistema a otro.

A veces usaremos el nombre: tipo String

Dominio

Secuencias de datos de tipo carácter, normalmente con restricciones de longitud dependientes del sistema.

Representación interna

Dependiente de cada sistema (se comentará a lo largo del curso).

Representación externa

Se escribirán entre comillas dobles. Por ejemplo: “Hola”, “Esto es una cadena, puede contener espacios y signos”, “n”, “el 3”, etc.

Operaciones

Los operadores binarios de relación entre cadenas vienen determinados por el orden generado por el código ASCII. Cuando se trate de palabras, equivale al orden lexicográfico (el del diccionario), salvo si se mezclan mayúsculas y minúsculas.

Aunque normalmente vamos a disponer de una gran cantidad de operadores para manejar cadenas, sólo vamos a dar como predefinidos los siguientes:

-**long**, que calcula la longitud de una cadena. Así, long (“hola”)=4

-**concat**, que construye una cadena uniendo otras dos. Por ejemplo,

concat(“Hola”, “ gente”)=”Hola gente”

PARTE III. Datos. Acción de asignación

1. Concepto de variable

Una **variable** se define como un dato cuyo **valor** puede cambiar dentro de un determinado **tipo de dato** y al cual se puede hacer referencia mediante un **nombre simbólico** o **identificador** de la variable. El tipo de dato y el identificador de una variable son dos **atributos invariables** asociados a ella. Otro atributo de una variable es su **ámbito** que es el lugar del programa en el que dicha variable es conocida y, por lo tanto, manipulable.

En nuestra notación algorítmica se **declararán todas las variables,** previamente a ser utilizadas. Para declarar una variable basta especificar el tipo de dato al que pertenecen sus valores potenciales y asignarle un nombre (identificador), por ejemplo:

variables

entero índice, contador

real medida, anchura

Respecto al ámbito de una variable se hablará más adelante, cuando se presenten los conceptos de algoritmo y subalgoritmo.

2. Concepto de constante

Una **constante** es un dato de **valor inalterable** durante la ejecución del algoritmo o programa.

Una constante se puede expresar en un programa de forma explícita, por ejemplo:

3.1416 'A ' 1000 lunes verdad

o bien de forma simbólica, asociándole previamente un nombre simbólico (identificador) e indicando su tipo, por ejemplo:

constante real pi = 3.1416

constante carácter letraA = 'A'

constante entero límite = 1000

Como ventaja de esta última solución cabe resaltar que se incrementa la **legibilidad** del programa y se facilitan sus posteriores **modificaciones** (el valor de la constante sólo debe modificarse en su definición y no en todos los lugares del programa donde aparece).

3. Asignación interna

Una variable declarada tiene, en principio, **valor indefinido.** La acción de asociar un valor a una variable se denomina **asignación.** El operador asignación lo representaremos de la forma " **=** " para distinguirlo del operador relacional de igualdad " == ". Una acción de asignación de valor a una variable se puede describir mediante una instrucción de asignación interna con la siguiente **sintaxis:**

```
variable = expresión
```

La semántica de la instrucción se puede describir en dos pasos:

1) Evaluar *expresión* obteniendo un valor

2) Asignar dicho valor resultante a *variable*

Ello trae como consecuencia que el valor resultante de evaluar *expresión* debe pertenecer al mismo tipo de dato que *variable*. En caso contrario se produce una **incompatibilidad** entre los operandos (*variable* y *expresión*) del operador asignación (=).

Las expresiones que aparecen en una instrucción de asignación pueden limitarse a simples valores constantes o variables, por ejemplo:

```
edad = 33;            letra = 'A';     peso= 77.50;
mismoPeso = peso;
```

No obstante, en el caso general, pueden aparecer en ellas una diversidad de operadores, por ejemplo:

```
x = x + 1
c = a*b + c/d
respuesta = (a < b + c) AND NOT (x > 10.5)
```

La siguiente pregunta que surge es la de ¿cómo evaluar una expresión en la que intervienen varios operadores?. La forma de proceder es similar a la conocida en matemáticas. A cada operador se le asigna un **nivel de prioridad.**

Mayor prioridad:	**NOT**					
	*****	**/**	**DIV**	**MOD**	**AND**	
	+	**-**	**OR**			
Menor prioridad:	**==**	**<**	**<=**	**>**	**>=**	**≠**

En caso de presentarse un **conflicto entre operadores** de un mismo nivel de prioridad, la expresión se evalúa de **izquierda a derecha**. El empleo de **paréntesis** permite alterar el orden de evaluación de la expresión.

4. Asignación externa (entrada de datos).

Puede ocurrir que un valor a tratar no esté disponible en el entorno de trabajo y deba ser introducido desde el exterior. Si este dato debe ser suministrado por un operador humano (mediante un teclado, desde un fichero previamente editado, etc.), éste lo hará utilizando una notación algorítmica (alfanumérica o textual) y no utilizando la misma representación interna que el ordenador.

Todo dato que sea introducido desde el exterior ("leido") se almacenará en una variable interna. Esta acción la realiza la instrucción **leer**. Como se verá más adelante corresponde a una llamada a ejecutar un procedimiento:

```
leer(variable)
```

Su funcionamiento (semántica) puede resumirse en los siguientes pasos:

1) Se busca el **tipo de dato** de *variable* (previamente ha debido ser declarada).

2) Se lee del dispositivo de entrada de datos una **secuencia de caracteres** cuya sintaxis corresponda a la representación en notación algorítmica alfanumérica de un dato del tipo asociado a *variable*.

3) Se realiza la **conversión** del valor leído de su representación alfanumérica a su representación interna equivalente.

4) Se **asigna** el valor leído y codificado a *variable*.

Vemos que la operación de lectura "introduce" un valor en una variable, es decir, realiza una asignación. Como el valor asignado proviene del exterior, hablaremos de **asignación externa**.

5. Salida de datos.

La instrucción básica de salida de datos es:

```
escribir(expresión)
```

Su **semántica** puede resumirse en los siguientes pasos:

1) Se **evalúa** *expresión* obteniendo un resultado

2) Se realiza la **conversión del** valor obtenido de su representación interna a su representación alfanumérica equivalente

3) Se **escribe** el valor resultante en su forma alfanumérica (secuencia de caracteres) por el dispositivo de salida de datos

Las diferentes acciones de salida de datos de un algoritmo van configurando un texto. Para facilitar su legibilidad es conveniente estructurarlo en líneas. La instrucción `escribirln()` fuerza el salto de línea después de la escritura.