Lenguajes de programación

Dos definiciones de lenguaje de programación podrían ser las siguientes:

> Notación para describir algoritmos de forma precisa.

> Conjunto de reglas, símbolos y palabras especiales usadas para construir un programa.

La *utilidad de un lenguaje* de programación es doble:

Especificar las acciones a ejecutar por el computador

Es una *palanca* que potencia la mente del programador facilitándole la resolución de cierto tipo de problemas y delimitando el mundo pensable.

Esta segunda característica ha provocado cierta especialización de los lenguajes.

Un recorrido por la historia de los lenguajes nos mostrará

que han surgido como respuesta a diferentes tipos de necesidades,

que la limitación en la construcción de programas está en el lado humano,

que la misma persona puede realizar programas más complejos o más grandes

- si el lenguaje exige la programación estructurada
- si el lenguaje es más expresivo
- si el lenguaje lo organiza todo con la orientación a objetos

PROGRAMACIÓN CABLEADA.

En la época de las primeras computadoras, cuando se deseaba ejecutar un algoritmo era necesario realizar un cableado que, una vez hecho, equivalía a un programa que la máquina tuviera inscrito en su estructura.

Cuando entre dos contactos (elementos del programa) se situaba un insecto (bug) produciendo una alteración en la estructura de las conexiones, el programa dejaba de realizar el trabajo que hasta entonces había hecho bien. Para corregir el programa, que ahora era otro distinto había que eliminar el insecto (debug). De esta época deriva las palabras inglesas «debug» con el significado de corregir un programa erróneo y «debuger» para nombrar al programa que ayuda en esta tarea de corrección de errores.

Naturalmente, esta forma de programar, por compleja, representaba un obstáculo para el desarrollo de máquinas y programas y, por ello, dio lugar a la aparición de otros tipos de lenguajes en que expresar los algoritmos.

MÁQUINA 1ª Generación

En el lenguaje nativo ó lenguaje máquina las instrucciones están constituidas por tiras de “unos” y “ceros” de diferente longitud donde una instrucción podría ser así:

10011 1000101110000100 1000011101110011		
10011	1000101110000100	1000011101110011
operación	1º oper y resultado	2º operando

Los 4 primeros dígitos binarios, bits, podían representar la operación, por ejemplo una suma, los 10 siguientes la dirección de memoria del primer sumando y del resultado y los últimos 10 la dirección de memoria del otro sumando. Esta se hubiera llamado una máquina de dos direcciones.

NOTA. Hubo máquina de cuatro direcciones cuando la memoria principal era un tambor que giraba a gran velocidad. Las direcciones eran de los dos operandos, del resultado y de la siguiente instrucción.

El lenguaje máquina es totalmente dependiente de esta, del diseño que tenga su unidad de control que, como sabemos, es la encargada de descodificar y hacer que se ejecuten las instrucciones.

> LENGUAJE MÁQUINA: la lengua que "entiende" la máqina.

Entre las ventajas del lenguaje máquina

no hay que traducirlo,

se ejecuta a mayor velocidad

ocupa poca memoria.

Entre los inconvenientes

codificación lenta y difícil,

largo periodo de puesta a punto del programa,

poca fiabilidad del código,

difícil verificación y

no portable: los programas escritos en él sólo se pueden ejecutar en la máquina de la que dependen o en las de la familia.

TRADUCCIÓN: Como la máquina solo entiende su lengua, los programas escritos en otro lenguaje hay que traducírselos antes de que los ejecute.

ENSAMBLADORES 2ª Generación

Un primer intento de superar los inconvenientes que presentaba el lenguaje máquina fue la utilización del ensamblador que es un lenguaje cercano a la máquina. Por ejemplo, la instrucción anterior en lenguaje máquina se escribe en un ensamblador hipotético así

```
ADD A, B
```

donde ADD es el mnemónico que recuerda la suma (addition) y A y B son variables que simbolizan posiciones de memoria que contendrán los sumandos y todavía no se ha decidido qué posiciones de memoria serán.

Las *diferencias con el lenguaje máquina* se pueden resumir en:

Uso de *símbolos* fáciles de recordar para representar la instrucción.

Uso de *variables simbólicas* en lugar de preocuparse por las direcciones de memoria.

Estas dos características hacen el código más legible, acortan los periodos de construcción de programas, ahorran esfuerzos en la codificación y se gana fiabilidad.

Sin embargo, el programa fuente escrito en ensamblador debe *traducirse* a código máquina para ejecutarlo. El programa que realiza la traducción y, eventualmente, convierte las variables simbólicas en direcciones de memoria reales se llama también *ensamblador* .

En comparación con el lenguaje máquina, se mantienen la velocidad de ejecución, la total dependencia de la máquina y la dificultad en la formación de programadores.

Actualmente, solo se emplea para programar cuestiones muy concretas como la capa más interna del sistema operativo, ciertas rutinas en las que la velocidad es una variable crítica, control de dispositivos electrónicos etc.

NOTA. Mientras que para los primeros ensambladores la relación entre sus instrucciones y las de la máquina es 1:1. en la actualidad se dispone de "macroensambladores" que son el resultado de la evolución de aquellos ganando en complejidad y potencia hasta tal punto que sus capacidades están muy cerca de las de lenguajes de alto nivel cercanos a la máquina como el lenguaje C.

ALTO NIVEL 3ª Generación

Todos sabemos que cuanto mayor sea la potencia de una herramienta mayor es el productividad de quien la usa, tanto en cantidad (produce más en el mismo tiempo) como en calidad (productos más complejos y elaborados).

Un lenguaje se puede ver como una herramienta cuyo diseño debería ayudar en la tarea de la programación. Para ello, debería

ser cercano al lenguaje natural para que el programador no tenga que usar fórmulas expresivas lejanas de las habituales.

incluir elementos del lenguaje matemático para hacerlo preciso.

La necesidad de acercar el lenguaje al programador dio como fruto la aparición de los lenguajes de alto nivel, es decir, más cercanos a la persona que en un rápido y muy corto resumen en orden temporal fueron

FORTRAN, (FORmula TRANslation) cuya primera versión fue publicada por IBM en 1956,

LISP (List Processing) fue creado por McCarthy en 1958. El primero en dar recursión, recolección de basura, funciones de primera clase y un lenguaje formal para su definición: el propio Lisp. Era ineficiente en sus primeras implantaciones y no tenía tipos.

ALGOL 60 (algorithmic language) 1958 es el fruto de una reunión que tuvo lugar ese año entre la americana ACM (Association for Computing Machinery) y la europea GAMM (Gessellchaft für Angewante Mathematick und Mechanik)

COBOL, (Comon Business-oriented Language), desarrollado por el comité CODASYL (Conference on Data Systems Languages) que se utilizó por primera vez en 1.960,

APL (A Programming Language) creado por Iverson a mitad de los 60 como un lenguaje matemático que después pasó a ser un

lenguaje de programación

ALGOL Wirth y Hoare reescriben el algol 60 en 1966 con algunos cambios. Es el antecesor directo de Pascal.

PASCAL, creado en 1968, revisado en 1972 y normalizado en 1.981. Hay dos normas estándar: el pascal básico y el ampliado.

C creado en 1972 por Dennis Ritchie como lenguaje para programar en Unix. En 1973 el propio Unix se reescribió en C. Es una evolución del Algol 60 a través del CPL (Combined Programming Language) de Strachey 1966, BCPL de 1969 y B 1970.

CPL --> BCPL --> B --> C

PROLOG 1972 por Alain Colmerauer de la Universidad de Marsella.

MODULA 2, desarrollado en 1983 después del PASCAL por el mismo autor Niklaus Wirth, que se proponía demostrar que un sistema operativo se puede escribir íntegramente en un lenguaje de alto nivel.

ADA, desarrollado a instancias del ministerio de defensa USA, cuya primera normalización data de 1983 por el ANSI.

C++ Stroustrup, 1986. Se le llama a veces «C con clases». Sus antecesores son C y Simula 67 en lo relativo a las clases.

Java, 1995, desarrollado por Sun Microsystems. Se parece a C++ pero tiene un modelo de objetos más sencillo.

Las ventajas de estos lenguajes con respecto a los anteriores

Proximidad al lenguaje humano (instrucciones como read, write, open, print),

- menor tiempo de formación de programadores,
- facilidad de puesta a punto y modificación de los programas

Portabilidad entre diferentes sistemas.

NOTA. Un programa es portable si se puede llevar a cualquier otra máquina, compilarlo sin ningún cambio y ejecutarlo en ella. Esta situación de portabilidad perfecta es rara pero no lo es que se puedan portar programas con pocos cambios.

NOTA. El programa ejecutable no es portable en general salvo a ordenadores de la misma 'familia' que tengan el mismo juego de instrucciones o un superconjunto de ellas.

Estos lenguajes no aprovechan tan eficientemente los recursos de la máquina como el ensamblador y máquina ya que gastan más memoria y tiempo de cómputo.

CLASIFICACIÓN DE LOS LENGUAJES

Una clasificación: la que surge de la estructura interna del lenguaje. Esta los divide en imperativos, los declarativos y otros de difícil encaje.

Imperativos

Los *imperativos* o procedimentales como Pascal, C, COBOL, FORTRAN...

Están influidos en su diseño por los detalles de la máquina.

Se basan en la idea de escribir las instrucciones a realizar por el ordenador como una receta de cocina, en ellos la unidad de trabajo es la instrucción.

Tienen 3 características:

Las *variables* denotan celdas de memoria de la máquina.

La *asignación* por la que se evalúa una expresión y el resultado se almacena en una posición de memoria. Esta instrucción hace nacer dos conceptos

- Valor i: lo que está a la *I* zquierda del operador de asignación. Denota una posición de memoria.
- Valor d: lo que está a la *D* erecha del operador de asignación. Denota el valor almacenado en una celda

Valor I <-- Valor D

La *iteración* es el medio para repetir secuencias de instrucciones. Es consecuencia de la arquitectura «Von Neuman» donde es la única forma de ejecutar varias veces un conjunto de instrucciones.

La instrucción de salto incondicional *goto* cuyo uso conduce a la construcción de programas desordenados e ininteligibles.

NOTA. Dijkstra criticó en su carta "Go to statment considered harmful" de 1968 dando lugar a la crisis del software que desembocó en la programación estructurada que dio claridad a los programas evitando la instrucción goto.

Declarativos

Los *declarativos* de diseño poco influido por la arquitectura de la máquina y muy basado, por el contrario, en un enfoque matemático de las descripciones. No disponen de instrucción de asignación. Se suele decir que tienen tres características:

«expresividad» que facilita la descripción de los problemas,

«fiabilidad» para proteger al usuario de errores y

«elegancia» desde el punto de vista matemático

Los lenguajes declarativos se agrupan en dos grandes tipos: funcionales y lógicos.

Declarativos: FUNCIONALES

Su origen está en el *problema de la computabilidad de funciones matemáticas* , en el llamado *cálculo lambda* .

Se caracterizan porque

Todas sus construcciones son funciones (aplicaciones) en el sentido matemático.

- Por eso se les llama lenguajes *aplicativos* .

Un programa en un lenguaje funcional no es más que una función que resulta de la *composición* de otras más simples y/o de primitivas, que aplicada a los datos de entrada produce la salida deseada.

Ejemplos son LISP, Scheme y ML que es como LISP con tipos.

Estos lenguajes son *referencialmente transparentes* el valor de una función depende exclusivamente de los valores de sus parámetros. Por ello, *no es posible la existencia de efectos laterales:*

No se modifican ni usan variables globales

No se modifican los parámetros.

En Pascal se puede aplicar este principio (salvo con archivos) sin mas que

No usar variables globales. ¡NO SON NECESARIAS!

No modificar parámetros de procedimientos o funciones.

El *almacenamiento y liberación de memoria es automático* .

Veamos el ejemplo del intérprete de LISP. (Lists Procesing)

El lenguaje maneja dos conceptos:

átomos como «a» o «juan» o «2008» o «udc»

listas como

- ( )
- (a)
- (a juan 2008 udc)
- ((a 2008) john (udc))

La primera es la lista vacía, la segunda es una lista que tiene un átomo, la tercera tiene cuatro átomos y la última es una lista que contiene listas y átomos.

LISP espera expresiones bien formadas para mostrar el resultado de forma inmediata, por ejemplo

```
> (+ 1 2 3 4)   {suma}
10
> (* n n )      {n ^2}
> (* 3 (* n n)) {3(n^2)}
```

Declarativos: LOGICOS

Lógicos . Trabajan con relaciones. Se basan en la idea de que programar con relaciones es más flexible que hacerlo con funciones ya que las relaciones tratan de manera uniforme argumentos y resultados.

PROLOG (*Pro* gramation et *Log* ique) es el más eximio de los lenguajes que intentan llevar a la práctica la programación lógica. Fue desarrollado Alain Colmerauer.

Una *relación* o *predicado* , es una tabla de varias filas y columnas. Por ejemplo, la relación esPadreDe sería una tabla de dos columnas como esta

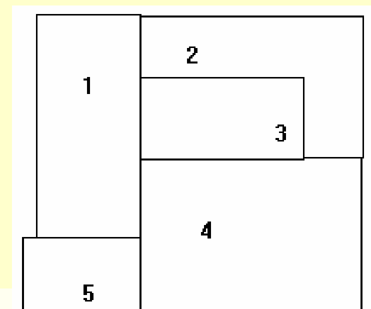
EsPadreDe	
Padres	**Hijos**
Juan	Carlos
Juan	Susana
Carlos	Pedro

Las relaciones se dan por medio de reglas del tipo *modus ponendo ponens* (si es verdad el antecedente entonces es verdad el consecuente) llamadas cláusulas de Horn que escriben primero el consecuente y luego el antecedente.

Un *hecho* es un caso especial de regla en la que no hay ninguna condición.

La programación lógica está dirigida por consultas sobre las relaciones. Una consulta simple consiste en preguntar si una tupla pertenece a una relación.

```
 {relación de adyacencia}
adyacente(1,2).
adyacente(2,1).
adyacente(1,3).
adyacente(3,1).
adyacente(1,4).
adyacente(4,1).
adyacente(1,5).
adyacente(5,1).
adyacente(2,3).
adyacente(3,2).
adyacente(2,4).
adyacente(4,2).
adyacente(3,4).
```



```
adyacente(4,3).
adyacente(4,5).
adyacente(5,4).
```

¿Es adyacente la región 2 a la 3?

```
?- adyacente(2,3),
yes
```

¿Es adyacente la región 5 a la 3?

```
?- adyacente(5, 3).
no
```

¿A qué regiones es adyacente la región 3?

```
?- adyacente(3, R).
R = 1 ;
R = 2 ;
R = 4 ;
no
```

Las respuestas son 1, 2 y 4 y ninguna otra.

Otros

Los *otros lenguajes* son difíciles de clasificar porque

o bien están a medio camino entre los imperativos y los declarativos como los lenguajes de flujo de datos o

no encajan en las clasificaciones anteriores.

Lenguajes *orientados a objetos*.

Pueden ser tanto imperativos como declarativos.

Su principal característica es la forma de manejar la información tratando de mantener la máxima cantidad de semántica del problema.

- Los imperativos pierden semántica por su adaptación a la máquina
- Los declarativos cambian semántica por fiabilidad.

En lo sucesivo hablaremos de lenguajes imperativos.

INSTRUCCIONES MÁS IMPORTANTES

Entre las instrucciones más importantes que podemos encontrar en un lenguaje de programación imperativo destacan las siguientes:

Asignación: es una acción que consiste en evaluar lo que hay a la derecha del símbolo de asignación, := en pascal, y el resultado de la evaluación se guarda en la posición de memoria representada por la variable que debe haber a la izquierda.

- variable := expresión;

Entrada y salida: son operaciones que permiten la transferencia de información con el exterior del programa a través de los dispositivos de entrada (teclado etc.) y salida (monitor etc.)

- escritura
  - write(argumento, argumento, argumento...);
  - writeln(argumento, argumento, ...)
  - writeln
- lectura
  - read(variable, variable, variable...);
  - readln(variable, variable, variable ...);
  - readln;

Control de flujo: son la excepciones al principio general de que "la siguiente instrucción a ejecutar es la está a continuación en memoria".

- *GOTO (PROHIBIDA)*
- Selección
  - SI .. FIN SI;
    - if (condición)
then sentencia_simple ;
    - if (condición)
then begin

```
        secuencia_de_sentencias
    end
```

- SI .. SI NO .. FIN SI;
  - ```
    if (condición)
    then sentencia
    else sentencia_simple;
    ```
  - ```
    if (condición)
    then begin
        secuencia_de_sentencias
    end
    else begin
        secuencia_de_sentencias
    end;
    ```
- SELECCIÓN MÚLTIPLE (CASE)

- Repetición

- MIENTRAS .. FIN MIENTRAS;
  - `while (condición) do sentencia_simple;`
  - ```
    while (condición) do begin
        secuencia_de_sentencias
    end;
    ```
- REPETIR .. HASTA;
  - ```
    repeat
        secuencia_de_sentencias
    until (condicion);
    ```
- DESDE .. HASTA
  - `for variable := valor_inicial to (condición) do sentencia_simple;`
  - ```
    for variable := valor_inicial to (condición) do begin
        secuencia_de_sentencias
    end;
    ```
  - `for variable := valor_inicial downto (condición) do sentencia_simple`
  - ```
    for variable := valor_inicial downto (condición) do begin
        secuencia_de_sentencias
    end;
    ```

- *Llamada y retorno de subrutinas* . Permiten dividir el programa en pequeñas unidades que serán ejecutadas en el momento que convenga

TIPOS DE SENTENCIAS EN PASCAL

Las sentencias en un programa escrito en Pascal representan las acciones que el algoritmo ha establecido. No debemos confundirlas con otros elementos que puede haber en el texto del programa tales como declaraciones o definiciones. Pueden ser simples y estructuradas.

Las *simples* realizan una de estas acciones:

asignación de un valor

activación de un procedimiento

goto : transferencia del control a otra sentencia

sentencia vacía : no hace nada

Las *estructuradas* : pueden ser

Compuestas : contienen una o más sentencias entre las palabras reservadas “BEGIN” y “END”.

Condicionales : controlan la elección de caminos alternativos IF, CASE

Repetitivas : especifican la repetición de sentencias WHILE, REPEAT, FOR

Sentencia *with* que simplifica el acceso a los datos de un registro y ahorra cálculos de dirección repetitivos en arrays de registros.