Tema 1. Algoritmos, programas y lenguajes de programación. (II)

PROGRAMAS

1. INTRODUCCIÓN.

En la primera parte del tema vimos que la resolución de un problema mediante un ordenador exige al menos los siguientes pasos:

1. Definición del problema.
2. Diseño del algoritmo.
3. Transformación del algoritmo en un programa.
4. Ejecución y validación del programa.

Obtener el algoritmo es el paso previo para escribir el programa. En la primera parte del tema vimos que un algoritmo es un conjunto de instrucciones para resolver un problema.

Para que el ordenador entienda estas instrucciones, el algoritmo ha de expresarse mediante un programa.

2. PROGRAMAS.

Un **programa** es un conjunto de instrucciones que se dan al ordenador para que realice un determinado trabajo.

Podemos decir que **programar** un ordenador es darle una serie de instrucciones para que realice una tarea concreta.

A la actividad de expresar un algoritmo en forma de programa se le denomina **Programación**.

Un programa se escribe utilizando un Lenguaje de programación.

3. LENGUAJES DE PROGRAMACIÓN.

Podemos definir un **lenguaje de programación** como un conjunto de símbolos e instrucciones que se usan siguiendo unas determinadas reglas para hacer que el ordenador realice una determinada tarea.

El propósito de un lenguaje de programación es permitir a las personas comunicarse con una computadora.

Un lenguaje de programación consta de:

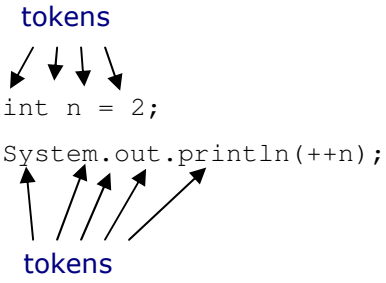
- Un **léxico**.
- Una **sintaxis**.
- Una **semántica**.

LEXICO

Conjunto de símbolos o tokens permitidos en el lenguaje.

Es el conjunto de componentes básicos e indivisibles que se pueden usar en el lenguaje: variables, literales, palabras reservadas, símbolos separadores, símbolos aritméticos,..etc.

Los componentes léxicos que forman el léxico de un lenguaje, se pueden asociar para formar estructuras: una asignación, una expresión,…etc. estructuras que pueden formar parte de un programa escrito en ese lenguaje.



`System.out.println` ⟶ Combinación de tokens

SINTAXIS

La sintaxis indica como combinar los símbolos o tokens para formar construcciones válidas del lenguaje.

Para definir la sintaxis de un lenguaje se utilizan las **gramáticas**.

Una gramática es un conjunto finito de reglas a través de las cuales se pueden generar el conjunto (por lo general infinito) de estructuras que forman el lenguaje.

`System.out.println` ⟶ Combinación sintácticamente correcta de tokens

`System.out_println` ⟶ Combinación sintácticamente incorrecta de tokens

SEMÁNTICA

Las construcciones deben de cumplir una serie de restricciones y condiciones: la compatibilidad de tipos en las operaciones, compatibilidad en asignaciones , etc.

La semántica es el conjunto de condiciones o restricciones que deben cumplir las estructuras.

La semántica determina el significado de cada construcción. Las construcciones deben ser semánticamente correctas es decir, deben indicar operaciones válidas y que tengan sentido dentro del lenguaje.

`System.out.println` ⟶ Combinación sintácticamente correcta de tokens

```
int n = 2;
```

`System.out.println(n%2.4);` ⟶ Combinación sintácticamente correcta
Semánticamente incorrecta, el operador % en Java no admite operandos de tipo double.

Las instrucciones básicas y comunes a casi todos los lenguajes de programación se pueden reunir en cuatro grupos:

- Instrucciones de **entrada/salida**. Instrucciones de transferencia de información y datos entre dispositivos periféricos (teclado, impresora, unidad de disco, etc.) y la memoria central.
- Instrucciones **aritmético-lógicas**. Instrucciones que ejecutan operaciones aritméticas (suma, resta, multiplicación, división) y lógicas (and, or, not, etc.).
- Instrucciones **selectivas**. Permiten la selección de tareas alternativas en función de los resultados de diferentes expresiones condicionales.
- Instrucciones **repetitivas**. Permiten la repetición de secuencias de instrucciones un número determinado de veces.

Aunque muchas veces se usan los términos 'lenguaje de programación' y 'lenguaje informático' como si fuesen sinónimos, no tiene por qué ser así, ya que los lenguajes informáticos engloban a los lenguajes de programación y a otros más, como, por ejemplo, el HTML (lenguaje para el marcado de páginas web que no es propiamente un lenguaje de programación).

Si se observan los lenguajes de alto nivel más habituales, vemos que estos lenguajes son un *pseudo inglés* reducido y con abundante aparato formal.

Normalmente nos encontraremos palabras como if, else, while, for, print, read, static, main, void, etc. También se utilizan signos como los paréntesis o las operaciones matemáticas como +, -, *, /.

Pero los procesadores solo son capaces de entender el lenguaje máquina, por lo tanto, todo programa escrito en otro lenguaje deberá ser traducido a lenguaje máquina para poder ejecutarse.

4. CÓDIGO FUENTE, CÓDIGO OBJETO Y CÓDIGO EJECUTABLE

Se llama **código fuente** al programa escrito en un lenguaje de programación determinado.

El código fuente se crea utilizando un editor de texto o un entorno de desarrollo determinado (por ejemplo NetBeans).

El lenguaje de programación utilizado para la redacción del código fuente es lo que se denomina un lenguaje de alto nivel, esto es, un lenguaje en que el programador puede expresarse con cierta facilidad por ser el más próximo al lenguaje natural.

Pero... hay un problema: **La máquina no lo entiende**

La máquina sólo comprende el lenguaje de máquina (código binario).

El código fuente necesita de un **proceso de traducción** que lo convierta a código máquina.

Comparando su actuación con la de un ser humano, un programa traductor equivale a un traductor profesional que, a partir de un texto, redacta otro nuevo traducido a otra lengua.

Aplicando el traductor sobre el código fuente se obtiene el **código objeto**.

Durante el proceso de traducción pueden utilizarse distintas herramientas: compiladores, intérpretes, máquina virtual, etc.

El programador escribe el código fuente con el lenguaje de programación elegido, y, en un proceso de traducción, este código fuente se convierte en código objeto.

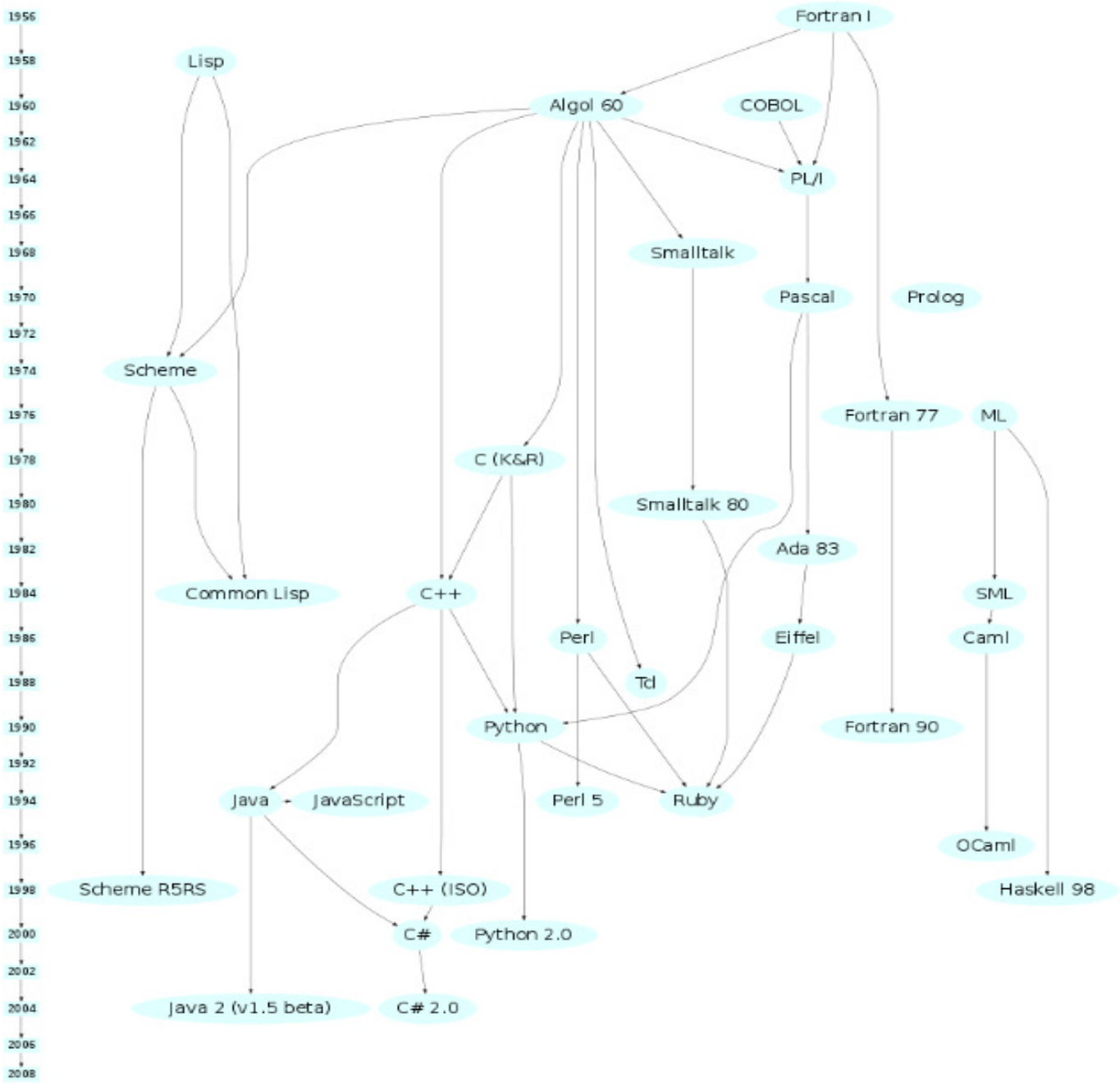
Si se trabaja en un entorno que disponga de librerías enlazables, el código objeto aún no es un programa ejecutable por el ordenador.

Deberá ser sometido a un proceso de enlazado, donde se añaden al programa las librerías necesarias.

El enlazador (linker) es el encargado de realizar este proceso. Con esta operación se incluyen en el programa determinadas librerías que son necesarias para que el programa pueda realizar ciertas tareas como, por ejemplo, manejar los dispositivos de entrada/salida de la máquina.

Al acabar este proceso se obtiene finalmente el **código ejecutable**.

4. EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN



4. CLASIFICACIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

Los lenguajes se pueden clasificar según distintos criterios:

4.1 SEGÚN SU NIVEL DE ABSTRACCIÓN

Los lenguajes de programación se pueden clasificar según su cercanía o lejanía al lenguaje natural. Los lenguajes cercanos al código máquina se llaman de bajo nivel y los más cercanos al lenguaje humano se llaman de alto nivel.

Según el nivel de abstracción se clasifican en:

- Lenguajes de Bajo Nivel:
  - Lenguaje Máquina.
  - Lenguajes Ensambladores.
- Lenguajes de Alto Nivel.

LENGUAJE MÁQUINA

El lenguaje máquina sólo se ha utilizado por los programadores en los inicios de la informática.

El lenguaje máquina es un lenguaje de Bajo Nivel cuyas instrucciones están formadas por cadenas de dígitos 0 y 1.

Sus instrucciones son cadenas binarias (series de dígitos 0 y 1) que especifican una operación y las direcciones de memoria de los datos implicados en la operación.

Ejemplo de código máquina:

```
0100  0010  0001  0000  0100  1101
0101  0100  0000  0010  0101  0111
0110  0011  0101  1001  0110  1001
```

Al lenguaje máquina también se le conoce como **código máquina ó código binario**.

La principal ventaja de programar en lenguaje máquina es la posibilidad de cargar el programa en memoria sin necesidad de traducirlo, lo que supone una velocidad de ejecución superior a cualquier otro lenguaje de programación.

Los inconvenientes son muchos y hacen que este tipo de lenguajes no sean recomendables para el programador de aplicaciones:

- El lenguaje máquina no es portable. Depende del hardware de la computadora, por tanto será diferente de una máquina a otra. Esto requiere hacer programas específicos para cada tipo de ordenador, de forma que si un programa debe ejecutarse en máquinas diferentes, habrá que codificarlo de maneras diferentes.
- Dificultad y lentitud en la codificación.
- Su conjunto de instrucciones es extremadamente irregular y por lo tanto es difícil recordarlo.
- Poca fiabilidad, es un método muy propenso a errores.
- Gran dificultad para verificar y poner a punto los programas.

LENGUAJE ENSAMBLADOR

Para evitar los lenguajes máquina, en los años 40 se creó un lenguaje simbólico que permitió no tener que programar en binario: el **lenguaje ensamblador** (assembly).

Es un lenguaje más fácil de utilizar que el lenguaje máquina y suponen el primer intento por acercar el lenguaje de los ordenadores al lenguaje natural.

El ensamblador utiliza símbolos nemotécnicos para las instrucciones, en lugar de las largas secuencias de ceros y unos que usa el lenguaje máquina.

Nemotécnicos típicos de operaciones aritméticas son: MOV, ADD, SUB, etc.

En este lenguaje cada instrucción equivale a una instrucción en código máquina.

Una instrucción típica de ensamblador es: ADD A, B

Esta instrucción indica que se sume el contenido de A con el de B y se guarde el resultado en A.

Como puede verse, es mucho más sencillo escribir esto que su equivalente código binario:

0110 0011 0101 1001 0110 1001

Ejemplo: Suma de 3 + 5 en un procesador 8086 (también Pentium, Pentium II, …)

Ensamblador Código máquina (Hexadecimal)

mov ax, 0003 B8 03 00

add ax, 0005 05 05 00

Ejemplo: programa que muestra el texto "Hola mundo" por pantalla:

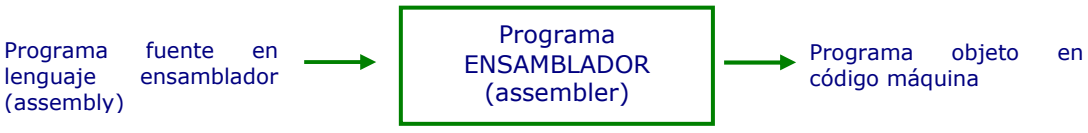
```
DATOS SEGMENT
   saludo db "Hola mundo!!!","$"
DATOS ENDS
CODE SEGMENT
   assume cs:code,ds:datos
START PROC
   mov ax,datos
   mov ds,ax
   mov dx,offset saludo
   mov ah,9
   int 21h
   mov ax,4C00h
   int 21h
START ENDP
CODE ENDS
END START
```

La principal utilización de este tipo de lenguajes es para programar los microprocesadores, utilizando el lenguaje ensamblador correspondiente a dicho procesador.

Como el único lenguaje que puede entender de forma directa el ordenador es el lenguaje máquina, un programa escrito en lenguaje ensamblador no puede ser ejecutado directamente, sino que requiere una fase de **traducción** al lenguaje máquina.

El programa original escrito en lenguaje ensamblador se denomina **programa fuente** y el programa traducido en lenguaje máquina se conoce como **programa objeto**.

El traductor de programas fuente en ensamblador a objeto es un programa llamado también ensamblador (**assembler**). No se debe confundir el ensamblador encargado de traducir el programa fuente con el lenguaje ensamblador.



La ventaja frente al lenguaje máquina es su mayor facilidad de codificación.

Los inconvenientes más notables son:

- Es un lenguaje no portable. Depende totalmente de la máquina y es distinto para máquinas distintas, igual que el lenguaje máquina.
- Dificultad para verificar y poner a punto los programas.
- La formación de los programadores es más compleja que la correspondiente a los programadores de alto nivel, ya que además de conocer las técnicas de programación, exige también el conocimiento de la arquitectura de la máquina.

LENGUAJES DE ALTO NIVEL

El ensamblador significó un gran avance respecto al lenguaje máquina aunque aún sigue siendo bastante complicado programar en este lenguaje. De hecho para hacer un programa sencillo requiere miles y miles de líneas de código.

Para evitar los problemas del ensamblador apareció la tercera generación de lenguajes de programación, la de los lenguajes de alto nivel.

Están diseñados para que las personas escriban y entiendan los programas de un modo más fácil que los lenguajes máquina y ensambladores.

Cada instrucción escrita en un lenguaje de alto nivel equivale a varias instrucciones en código máquina.

Tras varios intentos de representar lenguajes, en 1957 aparece el que se considera el primer lenguaje de alto nivel, el **FORTRAN** (**FORmula TRANslation**), lenguaje orientado a resolver fórmulas matemáticas. Por ejemplo un programa FORTRAN para escribir *Hola mundo* por pantalla es:

```
PROGRAM HOLA
   PRINT *, 'iHola, mundo!'
END
```

En 1960 se creó el **COBOL** como lenguaje de gestión.

En COBOL el programa *Hola mundo* sería:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HOLA.
PROCEDURE DIVISION.
DISPLAY "Hola mundo".
STOP RUN.
```

Presentan las siguientes ventajas respecto al lenguaje ensamblador:

- El tiempo de formación de los programadores es relativamente corto comparado con los lenguajes anteriores.
- La escritura de programas se basa en reglas sintácticas y palabras similares a los lenguajes humanos (read, write, print, open, …).
- Las modificaciones y puestas a punto de los programas son más fáciles.
- Reducción del coste de los programas.
- Portabilidad. Son independientes de la máquina, por tanto pueden ser ejecutados con poca o ninguna modificación en diferentes tipos de ordenadores.

Los inconvenientes son:

- Incremento del tiempo de puesta a punto, al necesitarse diferentes traducciones del programa fuente hasta conseguir el programa ejecutable definitivo en lenguaje máquina.
- No se aprovechan los recursos internos de la máquina, que se explotan mucho mejor en lenguajes máquina y ensambladores.
- Aumento de la ocupación de memoria.
- El tiempo de ejecución de los programas es mayor.

4.2 SEGÚN SU CAMPO DE APLICACIÓN

Según la utilización u orientación que posean la mayoría de las aplicaciones elaboradas, los lenguajes de programación se pueden clasificar en:

Lenguajes para Aplicaciones científicas.

Lenguajes para Aplicaciones de procesamiento de datos y gestión.

Lenguajes para Aplicaciones de tratamiento de textos.

Lenguajes para Aplicaciones en inteligencia artificial.

Lenguajes para Aplicaciones de programación de sistemas.

***Ejercicio**:*

Busca información sobre estos lenguajes indicando: objetivos, características de las aplicaciones, lenguajes más utilizados y sus *características más importantes*.

4.3 SEGÚN EL PARADIGMA DE PROGRAMACIÓN EMPLEADO

Un paradigma de programación representa un enfoque particular o filosofía para la construcción del software.

Los paradigmas de programación nos indican las diversas formas que, a lo largo de la evolución de los lenguajes, han sido aceptadas como estilos para programar y para resolver los problemas por medio de una computadora.

Los distintos paradigmas de programación han ido apareciendo por distintos motivos, principalmente para facilitar la tarea de programar según el tipo de problema a abordar y para facilitar el mantenimiento del software.

Los principales paradigmas de programación son:

- Paradigma imperativo o por procedimientos.
- Paradigma Declarativo.
- Paradigma orientado a objetos.

Algunos lenguajes solo soportan un paradigma de programación y otros permiten utilizar varios al mismo tiempo dando lugar a la programación multiparadigma.

PARADIGMA IMPERATIVO O POR PROCEDIMIENTOS.

Utilizado por los lenguajes de programación tradicionales no orientados a objetos como C, Pascal, Cobol, etc. Basan su funcionamiento en el concepto de procedimiento o función. Una función es un conjunto de instrucciones que reciben unos datos, operan sobre ellos y producen un resultado.

Podemos considerar los programas de este tipo como una sucesión de llamadas a funciones y procedimientos que manipulan datos pasados como parámetros.

En el siguiente ejemplo se puede ver el estilo de los programas procedurales: definir estructuras y tipos de datos y luego crear funciones que toman como parámetros variables de estos tipos y realizan distintas operaciones sobre ellos:

```
#include <iostream>
using namespace std;
```

```
struct Persona
{
       string nombre;
       string apellido;
       int edad;
};
void inicializar(Persona &, string, string, int);
bool es_mayor(Persona);
string nombre_completo(Persona);
int main(void)
{
    Persona p;
    inicializar(p, "Antonio", "Lozano", 30);
    cout << nombre_completo(p) << endl;
    if (es_mayor(p))
        cout << "Mayor de edad." << endl;
    else
        cout << "Menor de edad." << endl;
    system("pause");
}
void inicializar(Persona &p, string n, string a, int e)
{
    p.nombre = n;
    p.apellido = a;
    p.edad = e;
}
bool es_mayor(Persona p)
{
    return p.edad >= 18;
}
string nombre_completo(Persona p)
{
    return p.nombre + " " + p.apellido;
}
```

PARADIGMA DECLARATIVO.

Es el contrario del imperativo. Los programas se construyen señalando hechos, reglas, restricciones, ecuaciones, transformaciones y otras propiedades derivadas del conjunto de valores que configuran la solución. A partir de esta información el sistema debe de proporcionar un esquema que incluya el orden de evaluación que compute una solución.

En definida, se describe detalladamente el problema para que el sistema encuentre la solución.

Dentro de este paradigma existen dos **modelos de desarrollo**:

El modelo lógico. Los programas escritos en un lenguaje lógico están formados únicamente por expresiones lógicas, es decir, expresiones que solo toman como valor cierto o falso.

En los lenguajes imperativos las instrucciones se ejecutan normalmente en orden secuencial, una a continuación de otra, en el mismo orden en que están escritas que sólo varía cuando se alcanza una instrucción condicional o iterativa.

En el modelo lógico el orden de ejecución de las instrucciones no tiene nada que ver con el orden en que fueron escritas. Tampoco hay instrucciones de control propiamente dichas. Una instrucción se ejecutará automáticamente en cualquier momento en que se cumplan las condiciones especificadas.

El lenguaje más popular de este paradigma es el **PROLOG** aunque existen otros como Mercury y Oz.

Un ejemplo de programa escrito en PROLOG:

```
padrede('juan', 'maria'). % juan es padre de maria
padrede('pablo', 'juan'). % pablo es padre de juan
padrede('pablo', 'marcela').
padrede('carlos', 'ana').
hijode(A,B) :- padrede(B,A).
abuelode(A,B) :- padrede(A,C), padrede(C,B).
hermanode(A,B) :- padrede(C,A) , padrede(C,B), A \== B.
familiarde(A,B) :- padrede(A,B).
familiarde(A,B) :- hijode(A,B).
familiarde(A,B) :- hermanode(A,B).
?- hermanode('juan', 'marcela').
yes
?- hermanode('carlos', 'juan').
no
?- abuelode('pablo', 'maria').
yes
?- abuelode('maria', 'pablo').
no
```

El modelo funcional. Los programas están formados únicamente por funciones, entendiendo éstas no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas. Una función en estos lenguajes es una regla de correspondencia que asocia a cada elemento de un conjunto origen un elemento de un conjunto destino.

En este modelo la tarea se realiza evaluando composiciones funcionales. El resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que se produce el valor deseado. Una de sus características principales de este modelo es el empleo de la recursividad.

Ejemplos de este tipo de lenguajes son LISP, Scheme y Haskell.

Un ejemplo sencillo de programa escrito en LISP:

```
[1]> (defun que-es (n)
           (cond ((> n 0) 'positivo)
           ((< n 0) 'negativo)
           (T 'zero)))
QUE-ES
[2]> (que-es 5)
POSITIVO
[3]> (que-es -5)
NEGATIVO
[4]> (que-es 0)
ZERO
[5]> (que-es -10)
NEGATIVO
```

PARADIGMA ORIENTADO A OBJETOS.

La orientación a objetos se presenta como una alternativa al modelo tradicional por procedimientos, permitiendo crear sistemas resistentes al cambio y fáciles de mantener.

La diferencia fundamental entre la programación tradicional y la orientada a objetos está en la forma de tratar los datos y las acciones.

En la programación por procedimientos los datos y las acciones a realizar sobre ellos son cosas distintas. Se definen unas estructuras de datos y luego se definen por separado una serie de funciones que operan sobre ellas. Para cada estructura de datos se necesita un nuevo conjunto de funciones.

En la programación orientada a objetos los datos y las acciones a realizar sobre ellos están relacionados. Cuando definimos los datos (*objetos*) también definimos las acciones.

Podríamos ver un objeto como una estructura a la que le hemos añadido en su interior las funciones que determinan las acciones a realizar con el objeto.

En lugar de un conjunto de funciones que operan sobre unos datos tendremos una colección de objetos que interactúan intercambiando mensajes que transforman estados.

En la actualidad existen una gran cantidad de lenguajes orientados a objetos. Algunos de ellos son: C++, C#, VB.NET , Delphi, Java, PHP, Python, Ruby, Smalltalk, etc. Muchos de estos lenguajes no son puramente orientados a objetos, sino que son híbridos que combinan la POO con otros paradigmas.

El mismo ejemplo de programa por procedimientos visto al principio escrito ahora utilizando orientación a objetos en C++:

```
#include <iostream>
using namespace std;
class Persona
{
      private:
               string nombre;
               string apellido;
               int edad;
      public:
              void inicializar(string n, string a, int e)
              {
                   nombre = n;
                   apellido = a;
                   edad = e;
              }
              bool es_mayor()
              {
                   return edad >= 18;
              }
              string nombre_completo()
              {
                     return nombre + " " + apellido;
              }
};
int main(void)
{
    Persona p;
    p.inicializar("Juan", "Perez", 25);
    cout << p.nombre_completo() << endl;
    if (p.es_mayor())
        cout << "Mayor de edad." << endl;
    else
        cout << "Menor de edad." << endl;
    system("pause");
}
```

Evolución de los lenguajes de programación agrupados según el paradigma utilizado: