

1. Paradigmas de programación

- Desarrollo del software imposible si las tareas tuviesen que expresarse en código máquina.
- Lenguajes de más alto nivel: comprensibles, manejables y automáticamente convertibles a código máquina.
- Concentrar en las propiedades del problema que se pretende resolver, y no en registros, direcciones de memoria, ciclos máquina, ...

1.1. Perspectiva histórica

- Código máquina: depuración de errores (*debugging*) compleja
- Uso de dígitos (hexadecimales) para representar los códigos de operación -> Nemotécnicos

LD: Cargar registro

ST: Almacenar registro

R0, R1, ...: Nombres de los registros

156C	LD R5, PRECIO
166D	LD R6, IMPUESTOS
5056	ADDI R0, R5 R6
306E	ST R0, TOTAL
C000	HLT

Diseño del programa en **lenguaje ensamblador** y traducción automática a código máquina por el ensamblador.

Nueva generación de lenguajes de programación

1.1. Perspectiva histórica

- Primitivas del lenguaje: básicamente las mismas que el lenguaje máquina correspondiente.
 - Por tanto, dependencia total de la arquitectura de la máquina. No portabilidad.
 - Pensar en términos de lenguaje máquina y sus estructuras. Necesidad de primitivas de más alto nivel.
 - Lenguajes de **tercera generación**: FORTRAN y COBOL
 $\text{Total} = \text{precio} + \text{impuestos}$
 - Traductores: convierten un programa en código máquina.
 - Compiladores: Traducción global y posterior ejecución
 - Intérpretes: Traducción simultánea a la ejecución
- En general, se logra cuasi-independencia de la máquina, sin más que disponer del traductor adecuado.

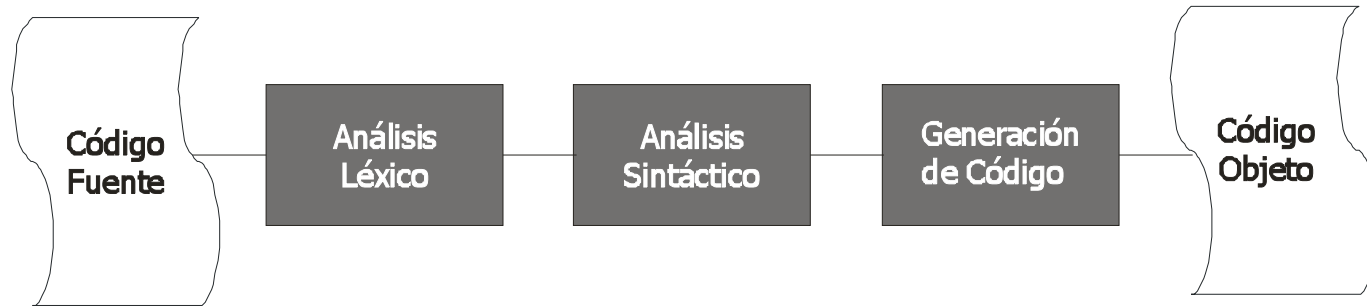
1.1. Perspectiva histórica

- Menor tiempo de formación del programador y de elaboración
- Depuración más sencilla
- Mayor consumo de recursos: memoria y tiempo de ejecución
- Estandarización de lenguajes más habituales: ANSI, ISO
- Entornos de programación “amigables”, lo más próximos posibles al lenguaje natural.
- Paradigmas de programación:

	Lisp	ML	Scheme		Funcional
		Simula	C++	Ada95	Orientada a Objetos
		Smalltalk	Visual Basic	Java	
Código	FORTRAN	BASIC	C	Ada	Imperativa
Máquina	COBOL	ALGOL	APL	Pascal	
		GPSS		Prolog	Declarativa

1.2. Proceso de traducción

- Traductores: toman el código fuente y generan el código objeto.



–Análisis léxico

- Reconocimiento de cadenas de símbolos: palabras clave, variables, constantes
- Asigna una marca a cada elemento reconocido

1.2. Proceso de traducción

–Análisis sintáctico

- “Ve” las marcas del A.L. e identifica la estructura gramatical del programa, reconociendo el papel de cada componente
- “signos de puntuación” para facilitar la identificación: { } ; ‘ ’ “ ” y palabras que identifican las diferentes estructuras sintácticas

```
if (expresión_lógica)
    sentencia ;
```

- Se genera la tabla de símbolos: variables utilizadas y su tipo. Genera mensajes de error en los lenguajes fuertemente tipados

```
total=precio+impuestos
```

–Generación de código

- Construcción de las sentencias en código máquina
- Optimización de código

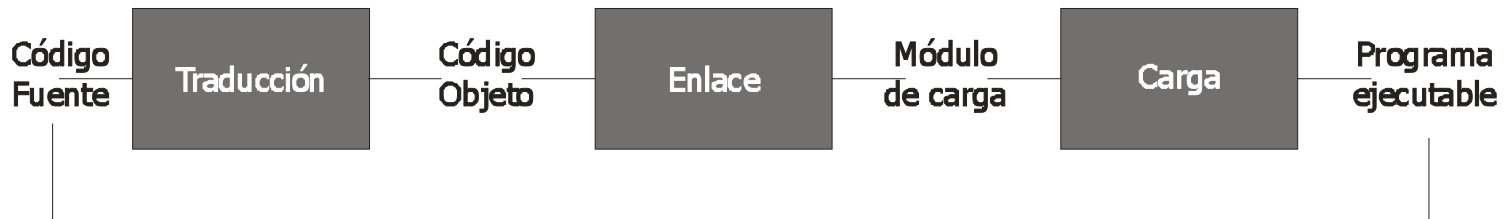
```
precioneto=precio-descuento
total=precioneto+impuestos
```

1.2. Proceso de traducción

Enlazado: posterior a la traducción y previo a la ejecución del programa.

- La traducción puede ser diferida.
- El código objeto contiene llamadas al S.O. o a otras aplicaciones.
- Deben “conectarse” los diferentes módulos y tareas pendientes.

Carga: traslado del código enlazado a memoria principal, desde donde se ejecuta.



Java: bytecode (código máquina “universal”) al que se traduce el código fuente.

Es ejecutado localmente por un intérprete.

1.3. Programación imperativa

- Programación imperativa o “*procedural*”: secuencia de comandos que, siguiendo un procedimiento, manejan los datos de entrada y generan el resultado.
 - Aproximación clásica: lenguajes alto/bajo nivel
 - Procedimiento: indicación paso a paso de las tareas a realizar para resolver el problema, traducida al lenguaje correspondiente.

1.4. Programación declarativa

- ¿Cómo se aplica en un sistema deductivo? Teniendo en cuenta que

$A \rightarrow B$ se puede interpretar como $\neg A \vee B$

Unificación: asignación de valores a variables

(Manolo está en X) \rightarrow (El coche de Manolo está en X)

Una instancia de la variable X puede ser X="el bar"

\neg (Manolo está en el bar) \vee (El coche de Manolo está en el bar)

que puede resolverse con, por ejemplo,

Manolo está en el bar

para dar

El coche de Manolo está en el bar

- PROLOG realizan resolución repetida sobre una colección de sentencias iniciales a las que se aplica razonamiento deductivo

1.4. Programación declarativa

- Predicado: representa un hecho

```
Vuelo(Madrid, Santiago).
```

```
Vuelo(Barcelona, Madrid).
```

- Reglas:

```
Vuelo(X, Z) :- (Vuelo(X,Y) AND Vuelo(Y,Z))
```

Procesado simbólico según el principio de resolución:

- Unificación (instancias de las reglas)
- Interactividad

permite confirmar objetivos como...

?Vuelo(Barcelona, Santiago)	¿V ó F?
?Vuelo(V, Madrid)	¿Instancias de V?
?Vuelo(Barcelona, W)	¿Instancias de W?
?Vuelo(V, W)	¿Instancias de V, W?

1.5. Programación funcional

- Programación funcional: las primitivas son funciones que se construyen para elaborar funciones más complejas que resuelvan el problema. Aproximación modular y reutilizable.

Ejemplo:

```
x+y          suma [x, y]
max2[x, y]   if x>y then
              x
              else
              y
max3[x, y, z] max2[max2[x, y], z]
```

- Las funciones manipulan (operan sobre y devuelven) listas de datos.

1.5. Programación funcional

- **Transparencia referencial:** Una función es RF o pura si no tiene *efectos laterales*:
 - No modifica parámetros
 - No usa variables globales
- **Idea “funcional” trasladada a la programación imperativa.**
 - En imperativa convencional, mucho más restringido el tipo de dato que se devuelve
 - No tanto en POO

1.6. Programación orientada a objetos

- Los datos son componentes individuales *activos* bien definidos con comunicaciones predefinidas.
 - Clases: plantillas para múltiples objetos, con características similares. Se agrupan en bibliotecas.
 - Objeto: instancias concretas de una clase
 - Atributos: características individuales
 - Variables de objeto
 - Métodos (comportamiento)
- Ejemplo: pequeño negocio con 20 empleados.

```
Class Ventana
{
    abrir ...
    Cerrar...
    Minimizar ...
    ...
}
Ventana V1;
```

- Programa: crear y destruir instancias de las clases

1.6. Programación orientada a objetos

- Propiedades

- Herencia: permite que una clase incluya a otras, pero también le permite añadir nuevas características particulares

```
Class NegociodeMensajería extends PequeñoNegocio
```

```
{
```

```
    ...Aquí se definirán los nuevos métodos para indicar cómo responderá cualquier elemento de la nueva clase ante nuevos.
```

```
    Además, hereda los métodos de la clase PequeñoNegocio...
```

```
}
```

```
PequeñoNegocio SupermercadoMarisa;
```

```
NegociodeMensajería DHL;
```

- Polimorfismo: una misma orden se “adapta” a los objetos (idea de sobrecarga de operadores aritméticos).

- Encapsulado: acceso restringido a los métodos y otras propiedades internas de un objeto (privados y públicos)

```
Class Ventana
```

```
{
```

```
    public ...
```

```
    private ...
```

```
    public...
```

```
}
```

1.6. Programación orientada a objetos

- Ventajas:
 - Construcción modular basada en objetos. Los datos quedan aislados entre sí por el encapsulamiento y no interfieren
 - Reutilización del código: los objetos son los bloques de partida para programas posteriores
 - Extensión sencilla, añadiendo nuevas clases
- Necesidad de un correcto diseño de las clases (generalización).
 - Metodologías de diseño (OOSE, UML, ...)
 - Software de apoyo al diseño
- Bibliotecas comerciales de clases:
interfases de usuario, aplicaciones bancarias, gestión BD, ...