

SEMÁNTICA

GRAMATICA

- Sintaxis
- Semántica
 - Semántica estática
 - Semántica dinámica
- Procesamiento de los programas
 - Intérpretes
 - Compiladores

SEMÁNTICA

La ***semántica*** describe el **significado** de los **símbolos, palabras y frases** de un **lenguaje** ya sea lenguaje natural o lenguaje informático **que es sintácticamente válido**

Para luego poder darle significado a una construcción del lenguaje

○ Ejemplos en C:

- `int vector [10];` (declaración)
- `if (a<b) max=a; else max=b;` (estructura de control)

SEMÁNTICA

• Ejemplo en C:

```
1  #include <stdio.h>
2  int x=9;
3  int Prueba()
4  {
5      int y;
6      y=x-1;
7      printf("%d\n",y);
8      return 0;
9  }
10 int Prueba1()
11 {
12     x=x+1;
13     printf("%d\n",x);
14     return 0;
15 }
16
17 int main()
18 {
19     Prueba();
20     Prueba1();
21     printf("Despues de la llamada a prueba1\n");
22     return 0;
23 }
```

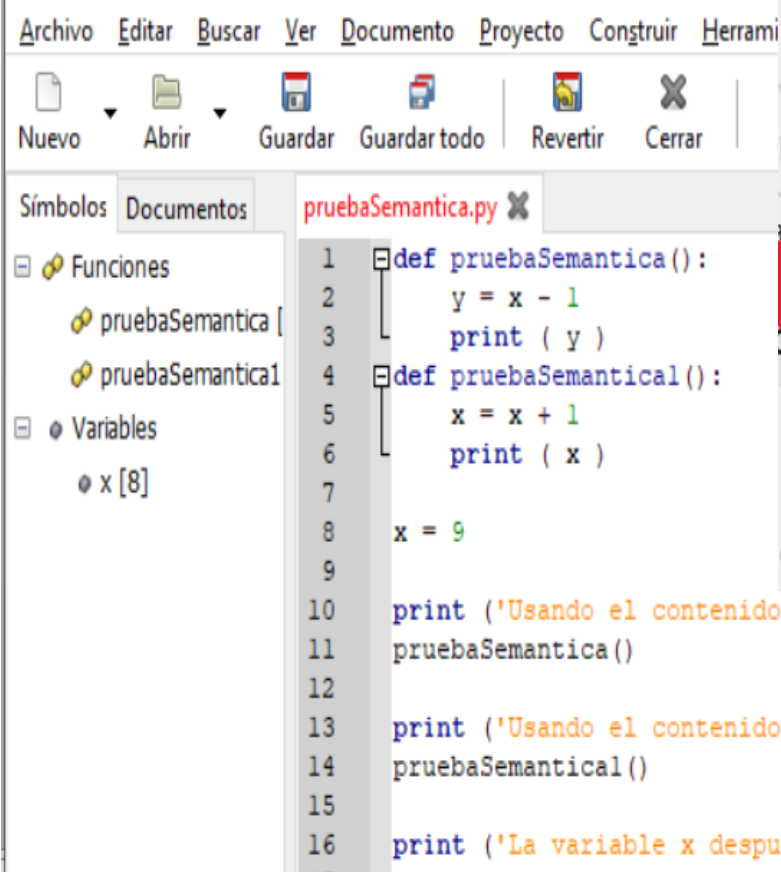
```
sh-4.3$ main
8
10
Despues de llamar a prueba
10
sh-4.3$
```

El significado en C

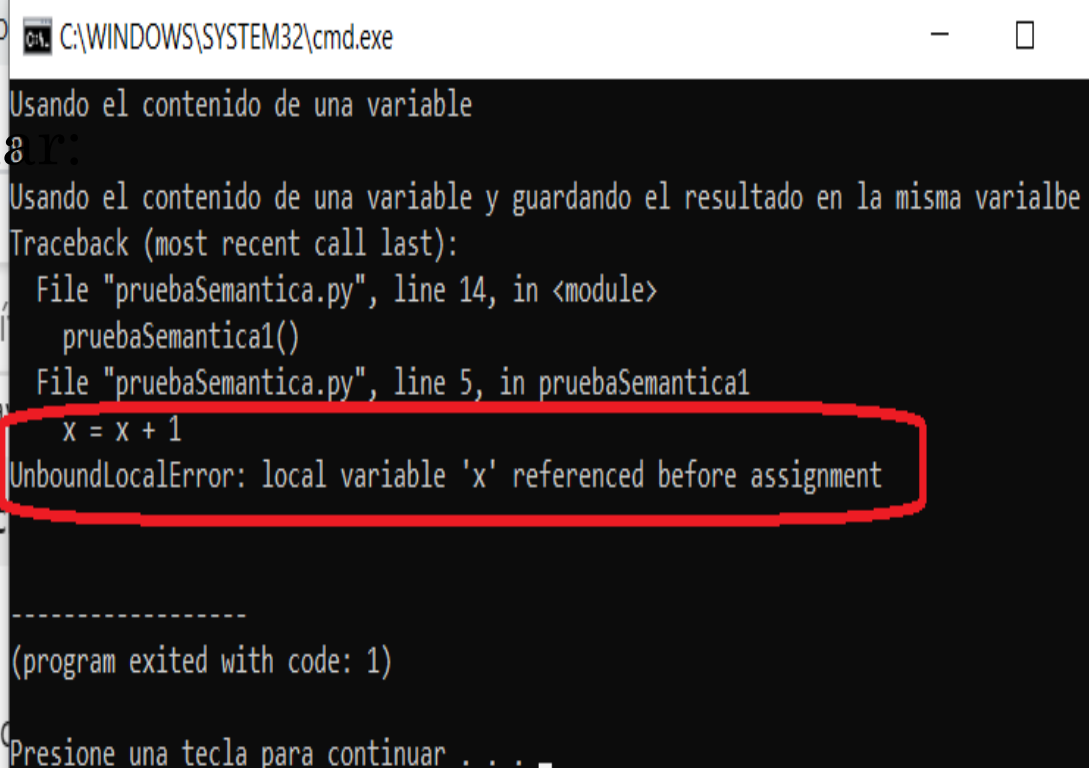
- Es **obligatorio** declarar variables y tipos.
- El alcance de la variable es **Global**. (El alcance de una variable es la parte del programa donde la variable es accesible...lo veremos mas adelante)
- Se ve que tanto **Prueba** como **Prueba1** usan la variable **x**. En un caso para **tomar** su valor. En el otro para **modificar** su valor.

SEMÁNTICA

• Ejemplo en Python similar a C:



```
1 def pruebaSemantica():
2     y = x - 1
3     print ( y )
4 def pruebaSemantica1():
5     x = x + 1
6     print ( x )
7
8 x = 9
9
10 print ('Usando el contenido de una variable')
11 pruebaSemantica()
12
13 print ('Usando el contenido de una variable y guardando el resultado en la misma variable')
14 pruebaSemantica1()
15
16 print ('La variable x después de las llamadas ', x)
```



```
C:\WINDOWS\SYSTEM32\cmd.exe
Usando el contenido de una variable
Usando el contenido de una variable y guardando el resultado en la misma variable
Traceback (most recent call last):
  File "pruebaSemantica.py", line 14, in <module>
    pruebaSemantica1()
  File "pruebaSemantica.py", line 5, in pruebaSemantica1
    x = x + 1
UnboundLocalError: local variable 'x' referenced before assignment

-----
(program exited with code: 1)
Presione una tecla para continuar . . .
```

¿Por qué el error?

La semántica en Python es muy distinta a C.

Python no es fuertemente tipado (no requiere declarar el tipo).

Desde el preciso momento en el que una **variable** recibe una **asignación** dentro de una **función**, pasa a ser calificada como **local** y tiene una visibilidad limitada al cuerpo de la función, deja de ser considerada como global si ya existía en el ámbito exterior. Hay que definirla global en el cuerpo de la función para que tome el valor global si necesitamos.

SEMÁNTICA - EJEMPLOS EN C CON 3 ERRORES DISTINTOS

1

```
main.c
1  #include <stdio.h>
2
3  int main()
4  {
5      int a, resul;
6      char cadena;
7
8      cadena = 'h';
9      resul = a + x;
10     printf(resul);
11     return 0;
12 }
13
```

2

```
main.c
1  #include <stdio.h>
2
3  int main()
4  {
5      int a, resul;
6      char cadena;
7
8      cadena = 'h';
9      resul = a + cadena;
10     printf(resul);
11     return 0;
12 }
```

3

```
main.c
1  #include <stdio.h>
2
3  int main()
4  {
5      int a;
6      int a, b;
7
8      b = a * 2;
9      printf(b);
10     return 0;
11 }
```

1. La variable x no fué declarada.
2. En una expresión se combinan diferentes tipos de datos y no hay reglas que lo permitan o lo resuelvan.
3. Se declararon 2 variables con el mismo nombre en un mismo entorno.

SEMÁNTICA

- ❑ **Hay que detectar otros errores semánticos**
 - ❑ **Hay características** de la estructura de los lenguajes de programación que son **difíciles** o **imposibles** de **describir** con **BNF/EBNF**
-
- **Tipos de semántica que analizaremos:**
 - Estática (antes de la ejecución)
 - Dinámica (durante la ejecución)

SEMÁNTICA

Semántica estática

- **No está relacionada con el significado de la ejecución del programa, está más relacionado con las formas válidas.**
- **El análisis está ubicado entre el análisis sintáctico y el análisis de semántica dinámica, pero más cercano a la sintaxis.**
- **Se las llama así porque el análisis para el chequeo se hace en compilación (antes de la ejecución).**

SEMÁNTICA

Semántica estática

- ¿Cómo detectar errores de compatibilidad de Tipos (ej. C)?
- ¿Cómo detectar errores de declaración de variables duplicadas (ej. C)?
- ¿Como detectar errores de variables no declaradas antes de referenciarlas (ej. Phyton)?
- **BNF/EBNF** no sirve para esto. Estas son gramáticas *libres de contexto* no se meten con el significado si con las formas)

"Mi perro canta una manta"

¿Es válido?

SEMÁNTICA

Semántica estática - Gramática de atributos

- Para describir la **sintaxis** y la **semántica estática** formalmente sirven las denominadas **gramáticas de atributos**, inventadas por Knuth en 1968.
- Son **gramáticas *sensibles al contexto* (GSC)**. Generalmente resuelven los aspectos de la **semántica estática**.
- La usan los **compiladores**

SEMÁNTICA

Semántica estática - Gramática de atributos

- A las **construcciones** del lenguaje se les **asocia información** a través de “**atributos**” **asociados** a los **símbolos** (terminales o no terminales) de la **gramática**
- Un **atributo** puede ser el **valor** de una **variable**, el **tipo** de una **variable o expresión**, **lugar** que ocupa una **variable en la memoria**, **dígitos significativos** de un **número**, etc.
- Los **valores de los atributos** se obtienen **mediante** las llamadas “**ecuaciones o reglas semánticas**” **asociadas a las producciones gramaticales**.

SEMÁNTICA

Semántica estática - Gramática de Atributos

- ❑ Las reglas sintácticas (producciones) son similares a BNF.
- ❑ Las ecuaciones (reglas semánticas) permiten detectar errores y obtener valores de atributos.
- ❑ Los atributos están directamente relacionados a los símbolos gramaticales (terminales y no terminales)
- ❑ Las GA se suelen expresar en forma tabular para obtener el valor del atributo

Regla gramatical	Reglas semánticas (<i>obtener el valor</i>)
------------------	---

Regla 1	Ecuaciones de atributo asociadas
<i>decl</i> → <i>tipo lista-var</i>	<i>lista-var.at = tipo.at</i>

Regla n	Ecuaciones de atributo asociadas
---------	----------------------------------

SEMÁNTICA

Semántica estática - Gramática de atributos

Como funciona la gramática de atributos:

- **Usa la tabla y machea si encuentra la producción/regla y del otro lado la ecuación que me permite llegar a los atributos.**
- Mira las **Reglas** (símil BNF/EBNF) y busca **atributos** para **terminales y no terminales**.
- Si encuentra el **atributo** debe llegar a obtener su **valor**.
- Para obtenerlo **genera ecuaciones**.

SEMÁNTICA- EJEMPLOS

Semántica estática - Gramática de atributos

- Ej. Gramática simple para una *declaración de variables* sólo de *tipo int* y *float* en el lenguaje C. con atributo *at*

Regla gramatical

decl → *tipo lista-var*

tipo → int

tipo → float

lista-var → *id*

*lista-var*₁ → *id*, *lista-var*₂

Similar a BNF pero no igual!

Cursiva NT

Normal T

-> se define como

, seguido

OR no existe, repetir fila

Reglas semánticas

lista-var.at = *tipo.at*

tipo.at = int

tipo.at = float

[*id.at* = *lista-var.at*

Añadetipo(id.entrada, lista-var.at)

[*id.at* = *lista-var.at*

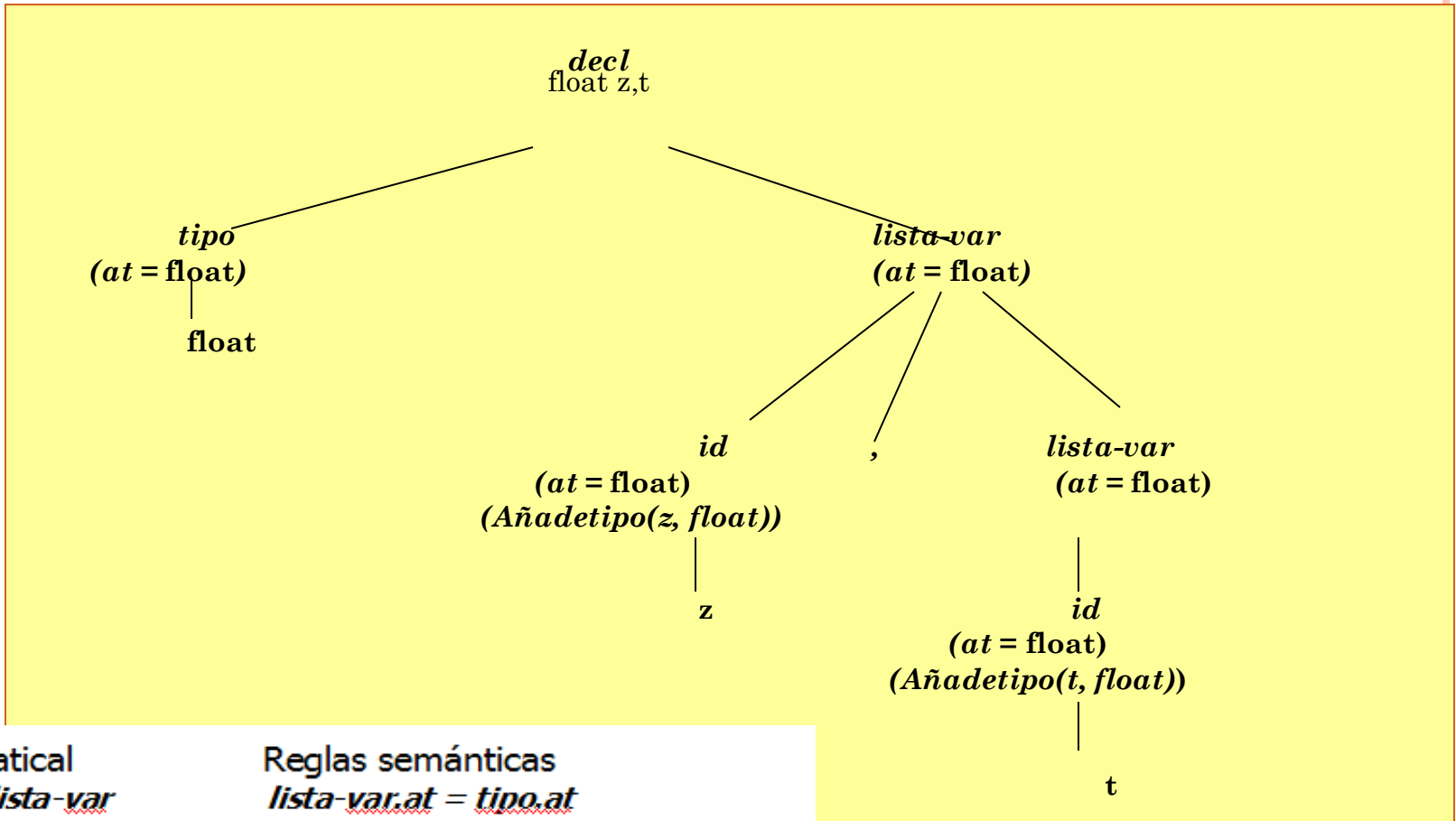
Añadetipo(id.entrada, lista-var₁.at)

[*lista-var*₂.at = *lista-var.at*

○ Arbol sintáctico atribuido

Parte de la gramática de atributos para ver si cumple

Ej. muestra los cálculos de atributo para la declaración: “float z,t”



Regla gramatical
 $decl \rightarrow tipo \ lista-var$

$tipo \rightarrow int$

$tipo \rightarrow float$

$lista-var \rightarrow id$

$lista-var_1 \rightarrow id, lista-var_2$

Reglas semánticas

$lista-var.at = tipo.at$

$tipo.at = int$

$tipo.at = float$

$id.at = lista-var.at$

$Añadirtipo(id.entrada, lista-var.at)$

$id.at = lista-var.at$

$Añadirtipo(id.entrada, lista-var_1.at)$

$lista-var_2.at = lista-var.at$

Esto permite la detección de errores cuando se agregan símbolos a la tabla de símbolos.

Si hay errores no pasa a ejecución.

SEMÁNTICA

Semántica estática - Gramática de atributos

De la ejecución de las ecuaciones:

- se ingresan **símbolos a la tabla de símbolos**,
- Detectar y dar **mensajes de error**
- detecta dos **variables iguales**,
- controla **tipo y variables de igual tipo**,
- ciertas **combinaciones no permitidas** (reglas específicas del lenguaje)
- otras **cosas no permitidas** por el lenguaje
- Es decir, el **chequeo de semántica estática**
- **Generar un código para el siguiente paso**

En definitiva, se debe tratar de representar todo lo que necesitamos que salte antes de la ejecución y no sea detectado sintácticamente.

SEMÁNTICA

Semántica dinámica:

- Es la que **describe** el **significado** de **ejecutar** las diferentes **construcciones** del lenguaje de **programación**.
- Su efecto se ve durante la **ejecución** del **programa**.
- Influirá la interacción con el **usuario** y **errores** de la programación

SEMÁNTICA

Semántica dinámica.

- Recordemos: Los programas **sólo se pueden ejecutar** si son **correctos** en la **sintáxis** y la **semántica estática**.

Analizar en C
`int notas[10];`

Un vector de 10 elementos consecutivos en memoria de tipo entero.

¿Qué pasa si exceden los límites del mismo? ¿O posición 10?

Si todo es correcto voy a obtener el valor del array de 10 elementos.

SEMÁNTICA

¿Cómo se describe la semántica dinámica?

- No es fácil escribirla
- No existen herramientas estándar (fáciles y claras) como en el caso de la sintáxis (diagramas sintácticos y BNF)
- Es complejo describir relación entre entrada y salida del programa
- Es complejo describir cómo se ejecutará en cierta plataforma.
- Etc.

SEMÁNTICA

Semántica Dinámica - soluciones más utilizadas:

- **Formales y complejas:**
 - Semántica **axiomática**
 - Semántica **denotacional**
- **No formal:**
 - Semántica **operacional**
- Sirven para **comprobar la ejecución, la exactitud de un lenguaje, comparar funcionalidades de distintos programas.**
- Se pueden **usar combinados**, no sirven todos para todos los tipos de lenguajes de programación

Veremos una descripción general para conocerlos

SEMÁNTICA

○ Semántica Axiomática

- **Considera al programa como “una máquina de estados” donde cada instrucción provoca un cambio de estado.**
- **Se parte de un axioma (verdad) que sirve para verificar "estados y condiciones" a probar**
- **Los constructores de un lenguaje de programación se formalizan describiendo como su ejecución provoca un cambio de estado (cada vez que se ejecuta).**
- **Se desarrolló para probar la corrección de los programas.**
- **La notación empleada es el “cálculo de predicados”**

SEMÁNTICA

○ Semántica Denotacional

- Se basa en la teoría de **funciones recursivas** y **modelos matemáticos**
- Se **diferencia** de la axiomática por la forma que describe los estados:
 - **Axiomática:** lo describe a través de los **PREDICADOS** (con **variables de estado**)
 - **Denotacional:** lo describe a través de **FUNCIONES** (funciones recursivas)

SEMÁNTICA

○ Semántica Denotacional

- Define una **correspondencia** entre los **constructores sintácticos** y sus **significados**
- Describe la **dependencia funcional** entre el **resultado** de la ejecución y sus **datos iniciales**
- Veamos un ejemplo.....

SEMÁNTICA DENOTACIONAL - EJEMPLO FUNCIÓN RECURSIVA

Producción definir binarios: $\langle \text{Nbin} \rangle ::= 0 \mid 1 \mid \langle \text{Nbin} \rangle 0 \mid \langle \text{Nbin} \rangle 1$

$\text{FNbin}(0)=0$ $\text{FNbin}(\langle \text{Nbin} \rangle 0) = 2 * \text{FNbin}(\langle \text{Nbin} \rangle)$

$\text{FNbin}(1)=1$ $\text{FNbin}(\langle \text{Nbin} \rangle 1) = 2 * \text{FNbin}(\langle \text{Nbin} \rangle) + 1$

Numero en binario **110**

El resultado es un 6

$\text{FNbin}(\langle \text{Nbin} \rangle \mathbf{0})$

$2 * \text{FNbin}(\langle \text{Nbin} \rangle \mathbf{1})$

$2 * [2 * \text{FNbin}(\mathbf{1}) + 1]$

$2 * [2 * 1 + 1]$

$2 * [3]$

6

resultado de las funciones

SEMÁNTICA

Semántica Operacional

- El **significado** de un **programa** se describe mediante **otro lenguaje de bajo nivel** implementado **sobre una máquina abstracta**
- **Los cambios** que se producen **en el estado** de la máquina abstracta, **cuando se ejecuta una sentencia** del lenguaje de programación, **definen su significado**
- Es un **método informal** porque se basa en **otro lenguaje de bajo nivel** y **puede llevar a errores**
- Es el **más utilizado** en los **libros** de texto para explicar el significado (PL/1 fue el primero)

Usa otro lenguaje para explicar el significado

SEMÁNTICA

Semántica Operacional

Ejemplo: en Pascal

Lenguajes

```
for i := pri to ul do  
begin  
.....  
end
```

Máquina abstracta

```
i := pri   (inicializo i)  
lazo if i > ul goto sal  
.....  
i := i + 1  
goto lazo  
sal .....
```

PROCESAMIENTO DE UN PROGRAMA

PROCESAMIENTO DE UN LENGUAJE

TRADUCCIÓN

- Las computadoras ejecutan un **lenguaje de bajo nivel** llamado “**lenguaje de máquina**” (con 0's y 1's)
- Un poco de historia...
 - Se Programaba en código de máquina (0's y 1's)
 - Esto era muy complejo y con muchos errores
 - Se pensaron soluciones

PROCESAMIENTO DE UN LENGUAJE TRADUCCIÓN

- Una **solución** fue reemplazar **repeticiones/ patrones de bits** por un **código**
- **Llamado código mnemotécnico** (abreviatura con el propósito de la instrucción).
- Así surgen:
 - “**Lenguaje Ensamblador**” que usaba estos códigos para programar
 - “**Programa Ensamblador**” que lo convertía a lenguaje de máquina

```
SUM #10, #11, #13  
SUM #13, #12, #13  
DIV #13, 3, #13  
FIN
```

PROCESAMIENTO DE UN LENGUAJE

TRADUCCIÓN - EJEMPLO HOLA MUNDO

Procesadores 80x86 de Intel

```
.data
msg:
.string "Hello, World!\n"
len:
.long . - msg
.text
.globl _start
_start:
push $len
push $msg
push $1
movl $0x4, %eax
call _syscall
addl $12, %esp
push $0
movl $0x1, %eax
call _syscall
_syscall:
int $0x80
ret
```

Procesadores de la familia Motorola 68000

```
start:
move.l #msg, -(a7)
move.w #9, -(a7)
trap #1
addq.l #6, a7
move.w #1, -(a7)
trap #1
addq.l #2, a7
clr -(a7)
trap #1
msg: dc.b "Hello,
World!", 10, 13, 0
```

- Cada máquina o familia de procesadores tiene su propio juego de instrucciones.

□ C/U su propio:

- Lenguaje ensamblador
- Programa ensamblador
- Código de máquina
- Código nemotécnico

PROCESAMIENTO DE UN LENGUAJE

TRADUCCIÓN - EJEMPLO HOLA MUNDO

- ❑ **Problemas que se producían:**
 - ❑ **imposible intercambiar programas entre distintas máquinas o de distintas familias de procesadores**
 - ❑ **diferentes versiones para una misma CPU pueden tener juegos de instrucciones incompatibles**
 - ❑ **modelos evolucionados de una familia de CPU pueden incorporar instrucciones nuevas**
- ❑ **Se buscaron otras soluciones**


PROCESAMIENTO DE UN LENGUAJE

TRADUCCIÓN

“Lenguajes de alto nivel” que permitieron esta abstracción

**Ejemplo de lenguaje máquina para el microprocesador 68000:
suma de dos enteros:**

Dirección	Código Binario	Código Ensamblador	Alto Nivel
\$1000	0011101000111000	MOVE.W \$1200,D5	Z=X+Y
\$1002	0001001000000000		
\$1004	1101101001111000	ADD.W \$1202,D5	
\$1006	0001001000000010		
\$1008	0011000111000101	MOVE.W \$D5,\$1204	
\$100A	0001001000000100		



PROCESAMIENTO DE UN LENGUAJE

TRADUCCIÓN: INTERPRETACIÓN Y COMPILACIÓN

- ¿Cómo los **programas escritos en lenguajes de alto nivel** pueden ser **ejecutados** sobre una computadora cuyo lenguaje es muy diferente y de muy **bajo nivel** que entiende 0s y 1s?
- Con **Programas Traductores** del lenguaje
 - Alternativas:
 - Interpretación
 - Compilación
 - Interpretación y Compilación (combinación)
- *No es decisión del programador.* El Programador sólo elige el lenguaje que más le convenga.
- *Es decisión del que crea el lenguaje*

INTERPRETACIÓN

- Hay un **Programa** escrito en **lenguaje de programación interpretado**
Ej: Lisp, Smalltalk, Basic, Python, Ruby, PHP, Perl, etc.
- Hay un **Programa** llamado **Intérprete** que realiza la traducción de ese lenguaje interpretado en el momento de ejecución

INTERPRETACIÓN

- El proceso que realiza cuando se ejecuta sobre cada una de las sentencias del programa es:
 - Leer
 - Analizar
 - Decodificar
 - Ejecutar

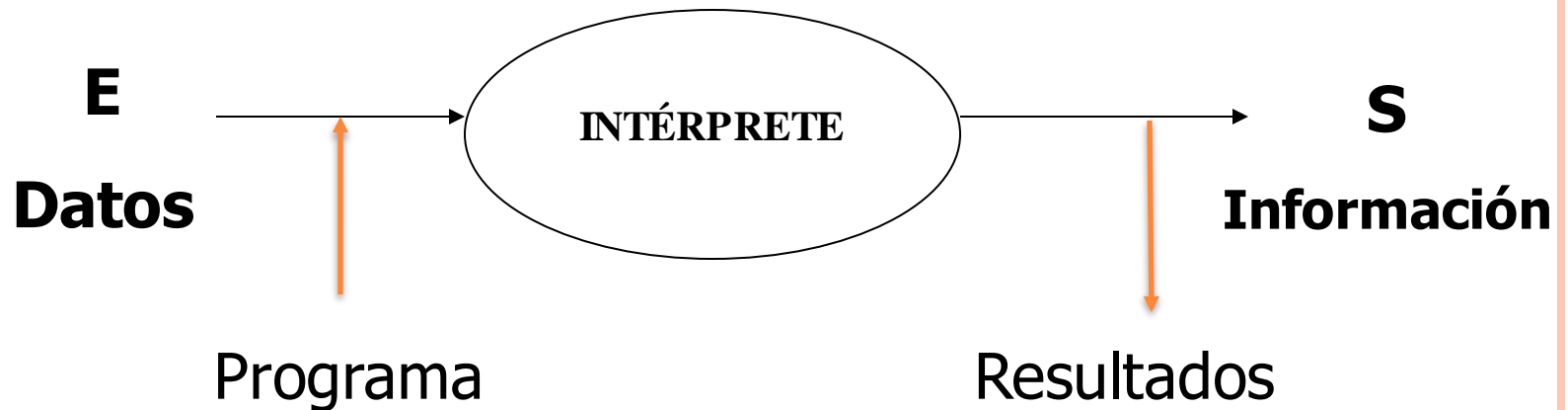
una a una las sentencias de programa)

 -
- Sólo **pasa por ciertas instrucciones** no por todas, según sea la ejecución (**ventajas y desventajas**)

INTERPRETACIÓN

- El **Intérprete** cuenta con una serie de **herramientas** *para la traducción a lenguaje de máquina*
 - Por **cada** posible **acción** hay un **subprograma** en **lenguaje de máquina** que ejecuta esa acción.
 - La interpretación se realiza **llamando** a estos **subprogramas** en la **secuencia adecuada** hasta **generar el resultado de la ejecución**.

INTERPRETACIÓN

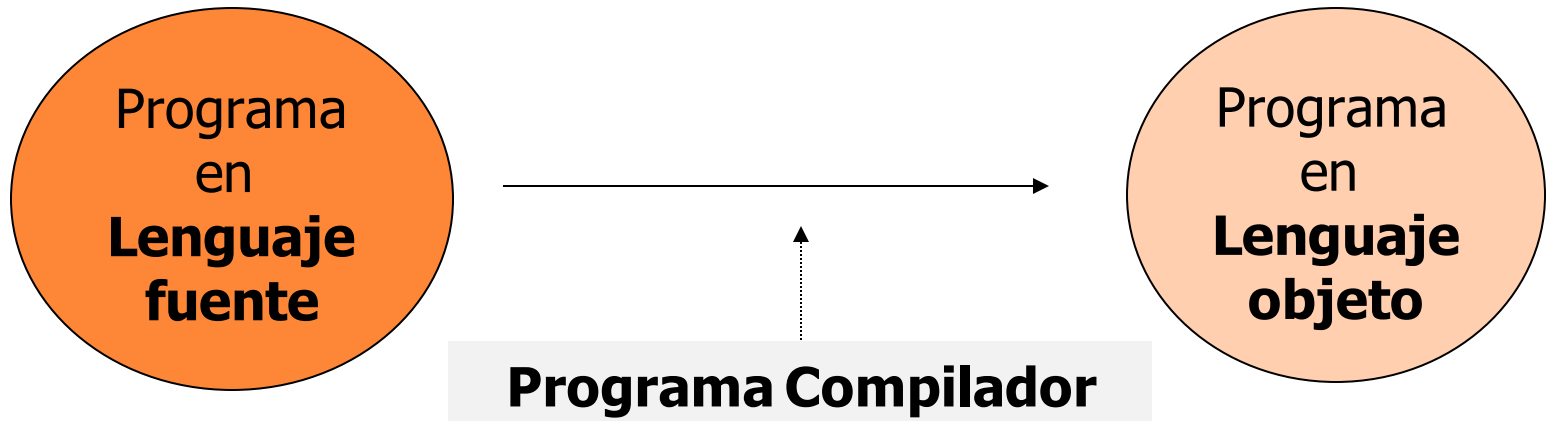


- Un intérprete ejecuta **repetidamente** la siguiente secuencia de acciones:
 - **Obtiene** la próxima sentencia
 - **Determina** la acción a ejecutar
 - **Ejecuta** la acción

COMPILACIÓN

- Elegimos un **lenguaje de alto nivel** de este tipo.
Ej.: Ada, C ++, Fortran, Pascal,.....
- Tenemos nuestro **programa** escrito
- Usamos un **programa** llamado **Compilador** que realiza la **traducción** a un **lenguaje de máquina**
- Se **traduce/compila** antes de ser **ejecutado**.
- Pasa por **todas** las instrucciones antes de la ejecución (**ventajas y desventajas**)
- El **código** que se genera se **guarda** y se puede **reusar** ya compilado.
- La compilación implica **varias etapas**, que analizaremos luego.....

COMPILACIÓN



- El compilador toma **todo** el programa escrito en un **lenguaje de alto nivel** que llamamos **lenguaje fuente** antes de su ejecución.
- Luego de la compilación va a generar un **lenguaje objeto** que es generalmente el **ejecutable (escrito en lenguaje de máquina)** o un **lenguaje de nivel intermedio (o lenguaje ensamblador)**.

TRADUCCIÓN

Comparación entre Compilador e Intérprete

- Por cómo se ejecuta:

- *Intérprete:*

- Ejecuta el programa de entrada directamente línea a línea
 - Dependerá de la acción del **usuario** o de la **entrada de datos** y de alguna **decisión del programa**
 - **Siempre se debe tener el Programa Intérprete**
 - **El programa fuente será público (necesito ambos)**

- *Compilador:*

- Se utiliza el compilador **antes de la ejecución**
 - Produce un **programa** equivalente en **lenguaje objeto**
 - **El programa fuente no será publico**

TRADUCCIÓN

Comparación entre Compilador e Intérprete

- Por el orden de ejecución:
 - *Intérprete:*
 - Sigue el **orden lógico** de ejecución (**no necesariamente recorre todo el código**)
 - *Compilador:*
 - Sigue el **orden físico** de las sentencias (**recorre todo**)

TRADUCCIÓN

Comparación entre Compilador e Intérprete (cont.)

○ Por el tiempo consumido de ejecución:

- *Intérprete:*
 - Por cada sentencia que pasa realiza el proceso de **decodificación (lee, analiza y ejecuta)** para determinar las operaciones y sus operandos. Es repetitivo
 - Si la sentencia está en un **proceso iterativo** (ej.: for/while), se realizará la tarea de decodificación **tantas veces como sea requerido**
 - La **velocidad** de proceso se puede ver **afectada**
- *Compilador:*
 - Pasa por **todas las sentencias**.
 - **No repite lazos**
 - Traduce todo de **una sola vez**.
 - Genera **código objeto ya compilado**.
 - La **velocidad de compilar** dependerá del **tamaño del código**

TRADUCCIÓN

Comparación entre Compilador e Intérprete (cont.)

○ Por la eficiencia posterior:

• *Intérprete:*

- Más lento en ejecución. Se repite el proceso cada vez que se ejecuta el programa
- Para ser ejecutado en otra máquina se necesita tener si o si el intérprete instalado
- El programa fuente será publico

• *Compilador:*

- Más rápido ejecutar desde el punto de vista del hardware porque ya está en un lenguaje de más bajo nivel.
- Detectó más errores al pasar por todas las sentencias
- Está listo para ser ejecutado. Ya compilado es más eficiente.
- Por ahí tardó más en compilar porque se verifica todo previamente
- El programa fuente no será publico

TRADUCCIÓN

Comparación entre Compilador e Intérprete (cont.)

- **Por el espacio ocupado:**

- ***Intérprete:***

- ❖ **No pasa por todas las sentencias** entonces ocupa **menos espacio de memoria.**
 - ❖ **Cada sentencia se deja en la forma original** y las **instrucciones interpretadas** necesarias para ejecutarlas **se almacenan** en los subprogramas del intérprete **en memoria**
 - ❖ Cosas como **tablas de símbolos, variables y otros** se **generan** cuando se usan en forma más dinámica

- ***Compilador:***

- **Si pasa por todas las sentencias**
 - **Una sentencia puede ocupar decenas o centenas de sentencias de máquina al pasar a código objeto**
 - Cosas como **tablas de símbolos, variables, etc.** **se generan siempre se usen o no**
 - Entonces el compilador hace **ocupar más espacio**

TRADUCCIÓN

Comparación entre Compilador e Intérprete (cont.)

- **Por la detección de errores:**

- ***Intérprete:***

- Las **sentencias del código** fuente pueden ser **relacionadas** directamente con la **sentencia en ejecución** entonces **se puede ubicar el error**.
 - **Es más fácil detectarlos** por donde pasa la ejecución
 - **Es más fácil corregirlos**

- ***Compilador:***

- Se pierde la **referencia** entre el **código fuente** y el **código objeto**.
 - Es **casi imposible ubicar el error**, pobres en significado para el programador.
 - Se deben **usar otras técnicas** (ej. **Semántica Dinámica**)

TRADUCCIÓN

Combinación de Técnicas de Traducción:

- *Primero interpreto y luego compilo*
- *Primero compilo y luego interpreto*

TRADUCCIÓN

Combinación de Técnicas:

- **Interpretación pura y Compilación pura** son dos extremos
- En la práctica **muchos lenguajes combinan ambas técnicas** para sacar provecho a cada una
- Los compiladores y los intérpretes **se diferencian** en como **reportan los errores de ejecución**,
- También, hay **otras diferencias** que vimos anteriormente en la **Comparación**
- **Ciertos *entornos de programación*** contienen las dos versiones: **interpretación y compilación.**

TRADUCCIÓN

1- Primero Interpreto y luego Compilo

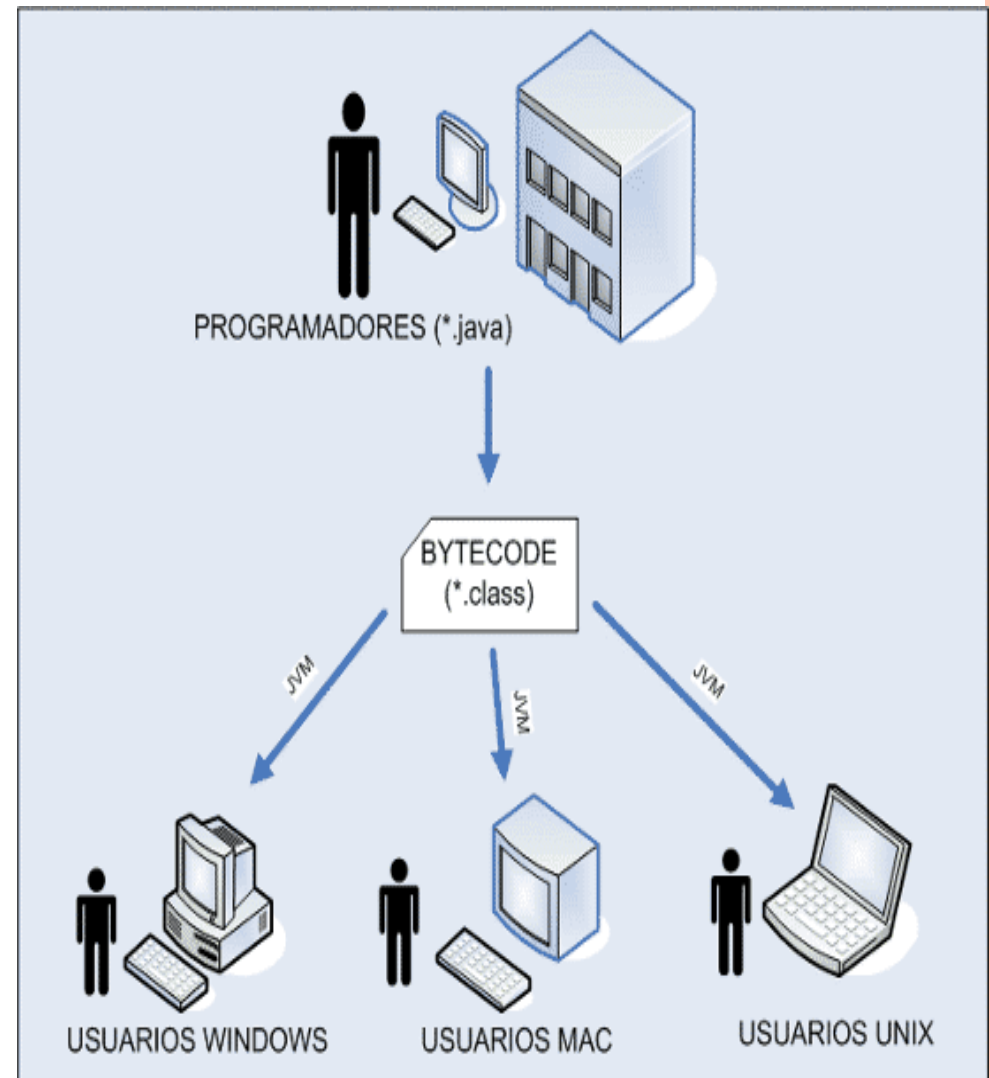
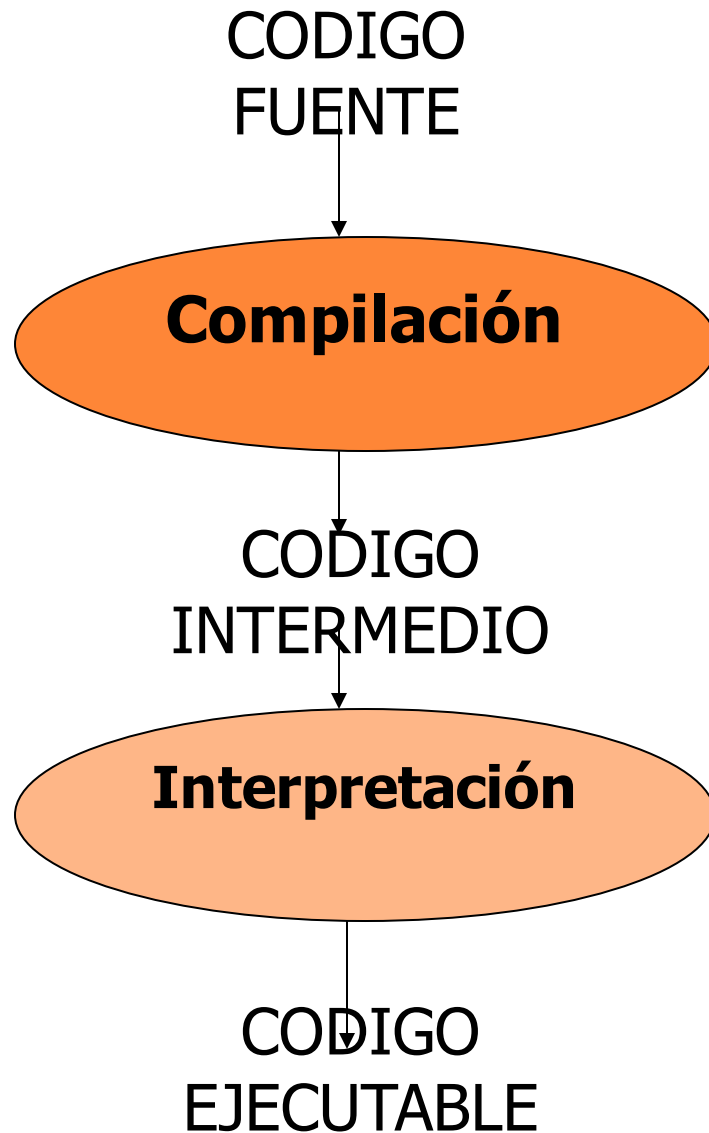
- Se utiliza el **intérprete** en la **etapa de desarrollo** para facilitar el **diagnóstico de errores**.
- Con el **programa validado** se **compila** para generar un **código objeto más eficiente**.

TRADUCCIÓN

2- Primero Compilo y luego Interpreto

- Se hace traducción a un código intermedio a bajo nivel que luego se interpretará
- ▣ Sirve para generar **código portable**, es decir, código fácil de transferir a diferentes máquinas y con diferentes arquitecturas.
- **Ejemplos:**
 - **Compilador Java genera código intermedio** llamado “**bytecodes**”. Luego es **interpretado** por **JavaScript** en la **máquina cliente**.
 - **C# (Cii Sharp) de Microsoft .NET**
 - **VB.NET (Visual Basic .NET de Microsoft)**
 - **PHYTON, etc.**

TRADUCCIÓN: COMBINACIÓN DE ESTAS TÉCNICAS



ESQUEMA COMPILADO – INTERPRETADO JAVA

COMPILADORES – CÓMO FUNCIONAN

- **Traducen todo el programa**
- Pueden **generar un "código ejecutable"** (.exe) o un **"código intermedio"** (.obj)
 - **Ej. de programas que se compilan:**
 - **C, Ada, Pascal, etc.**

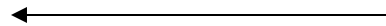
**La compilación puede ejecutarse
en 1 o 2 etapas**

COMPILADORES – CÓMO FUNCIONAN

- En **ambos** casos se **cumplen varias sub-etapas**, las principales son:

- 1) Etapa de Análisis

- Análisis léxico (Programa Scanner)
- Análisis sintáctico (Programa Parser)
- Análisis semántico (P. Semántica estática)



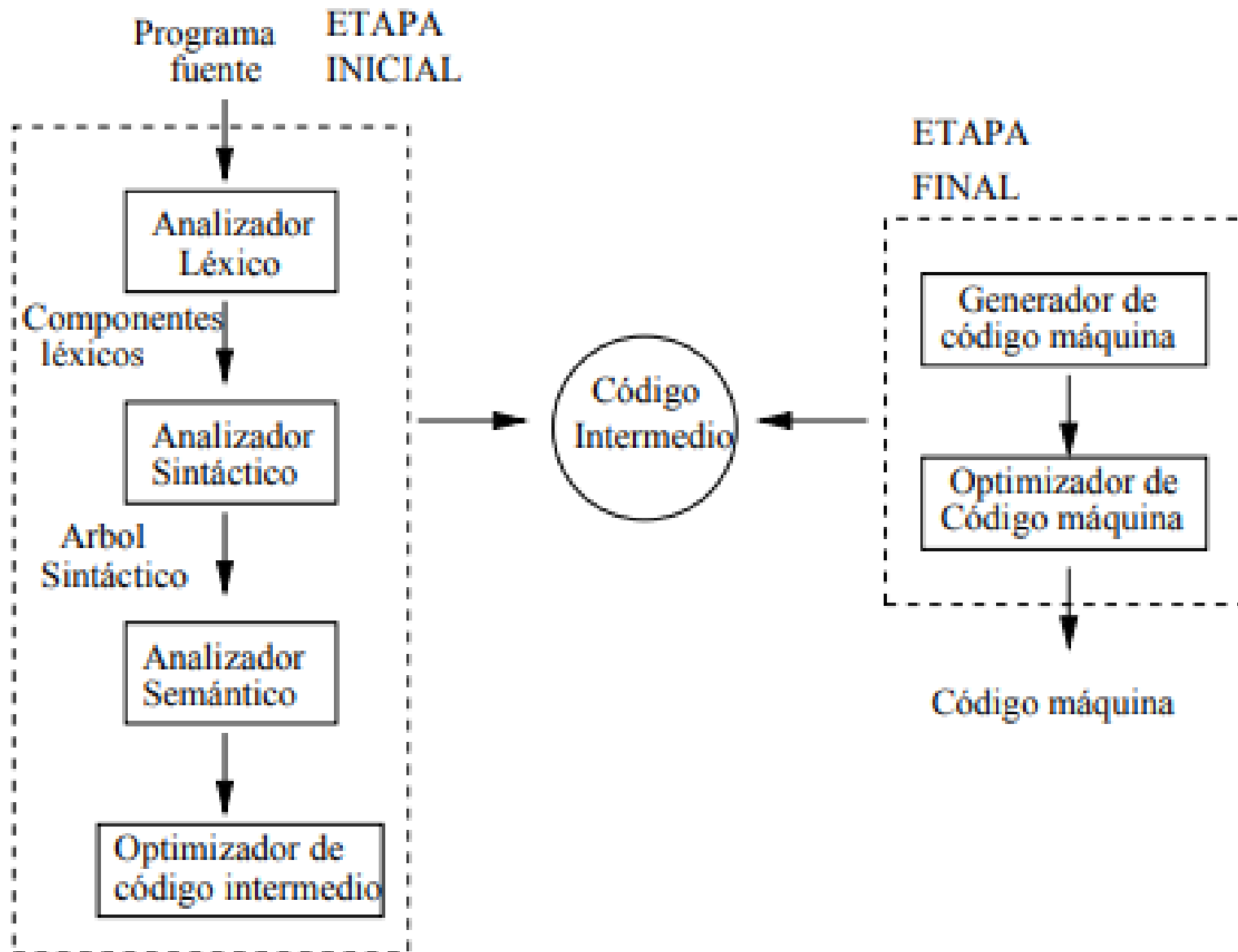
- 2) Etapa de Síntesis

- Optimización del código
- Generación del código

***Se puede generar
código intermedio***

- 1) más vinculado al código fuente
- 2) más vinculado a características del código objeto y del hardware y arquitectura

COMPILADORES – CÓMO FUNCIONAN



COMPILADORES

- 1) Etapa de Análisis del programa fuente
 - Análisis léxico (Scanner):
 - Es el que lleva más tiempo
 - Hace el análisis a nivel de palabra (LEXEMA)
 - Divide el programa en sus elementos: *identificadores, delimitadores, símbolos especiales, números, palabras clave, palabras reservadas, comentarios*, etc.
 - Analiza el tipo de cada TOKEN para ver si son válidos
 - Filtra comentarios y separadores como: espacios en blanco, tabulaciones, etc.

COMPILADORES

- **Etapas de Análisis del programa fuente**
 - **Análisis léxico (Scanner) cont:**
 - **Genera errores si la entrada no coincide con ninguna categoría léxica**
 - **Convierte a representación interna los números en punto fijo o punto flotante**
 - **Pone los identificadores en la tabla de símbolos**
 - **Reemplaza cada símbolo por su entrada en la tabla de símbolos**
 - **El resultado de este paso será el descubrimiento de los items léxicos o tokens.**

COMPILADORES

- Ejemplo de token, lexema y regla

ID	Componente léxico	Lexema	Patrón
201	IF	if	if
202	OP_RELACIONAL	<, >, !=, ==	<, >, !=, ==
203	IDENTIFICADOR	var, s1, suma, prom	letra (letra dígito gb)*
204	ENTERO	2, 23, 5124	(dígito)+

COMPILADORES

- **Etapas de Análisis del programa fuente**
 - **Análisis léxico (Scanner):**

```
x:=a+b*c;  
y:=3+b*c;
```

Analizador
Léxico

TOKENS

(id, "x")	(op, ":=")	(id, "a")
(op, "+")	(id, "b")	(op, "*")
(id, "c")	(punt, ";")	
(id, "y")	(op, ":=")	(num, "3")
(op, "+")	(id, "b")	(op, "*")
(id, "c")	(punt, ";")	

COMPILADORES

- **Análisis sintáctico (Parser):**
 - El análisis se realiza a **nivel de sentencia**.
 - Se **identifican las estructuras; *sentencias, declaraciones, expresiones, etc. ayudándose*** con los tokens.
 - El **analizador sintáctico** se alterna con el **análisis lexico y semántico**. Usualmente se utilizan técnicas basadas en **gramáticas formales**.
 - Se usa una **gramática para construir el "árbol sintáctico" del programa**.
 - Toma una **sentencia** y se **compara con el "árbol derivación"** para ver si lo que entra es correcto

COMPILADORES

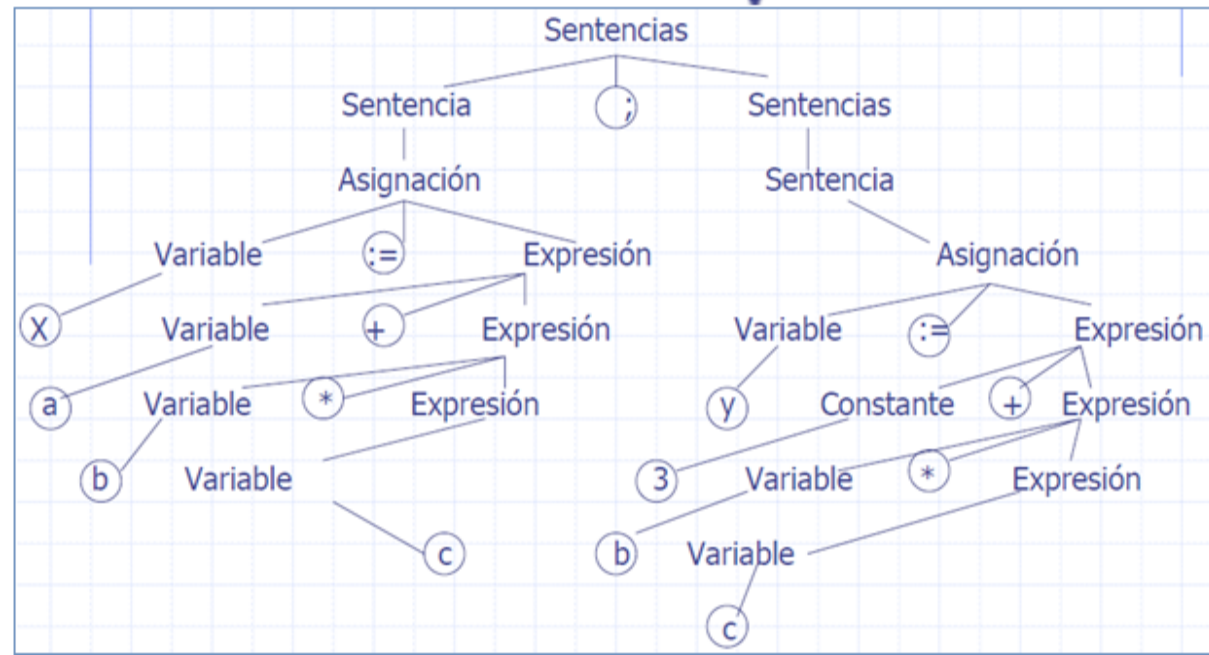
○ Análisis sintáctico (Parser):

x := a + b * c;
y := 3 + b * c;



TOKENS		
(id, "x")	(op, ":=")	(id, "a")
(op, "+")	(id, "b")	(op, "*")
(id, "c")	(punt, ";")	
(id, "y")	(op, ":=")	(num, "3")
(op, "+")	(id, "b")	(op, "*")
(id, "c")	(punt, ";")	

Analizador
Sintáctico



COMPILADORES

Análisis semántica (semántica estática):

- **Fase medular, una de las más importantes**
- **Las estructuras sintácticas reconocidas por el analizador sintáctico son procesadas y la estructura del código ejecutable continúa tomando forma.**
- **Se agrega otro tipo de información implícita (como variables no declaradas)**

COMPILADORES

Análisis semántica (semántica estática):

- Se realiza la comprobación de tipos (aplica gramática de atributos)
- Se agrega a la tabla de símbolos los descriptores de tipos
- Se realizan comprobaciones de duplicados, problema de tipos, etc.
- Se realizan comprobaciones de nombres. (ej: toda **variable** debe estar **declarada** en su entorno)
- Es el **nexo** entre etapas inicial y final del compilador (Análisis y Síntesis)

COMPILADORES

Generación de código intermedio:

- **Transformación del código fuente en una representación de código intermedio para una máquina abstracta.**
- **A esta representación intermedia, que se parece al código objeto pero que sigue siendo independiente de la máquina, se le llama código intermedio.**

El código objeto es dependiente de la máquina

COMPILADORES

Generación de código intermedio:

- Debe ser **fácil de producir**
- Debe ser **fácil de traducir al programa objeto**
- Hay **varias técnicas**
- El código intermedio más habitual es el **código de 3-direcciones**.
- Pasa todo el código a secuencia de **proposiciones** de la forma **$x := y \text{ op } z$**

COMPILADORES

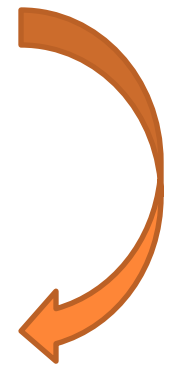
Ejemplo de generación de código intermedio:

- **Código de 3-direcciones** Forma: $A := B \text{ op } C$,
- A, B, C son **operandos/variables**
- **op** es un **operador binario**
- Se permiten **condicionales simples**
- Se permiten **saltos**.
- Cada **sentencia** se traduce en **N líneas**

while (a >0) and (b < (a*4-5)) do a:=b*a-10;

```
L1: if (a>0) goto L2
    goto L3
L2: t1:=a*4
    t2:=t1-5
    if (b < t2) goto L4
    goto L3
```

```
L4: t1:=b*a
    t2:=t1-10
    a:=t2
    goto L1
L3: .....
```



COMPILADORES

- **2) Etapa de Síntesis:**

- **Construye el programa ejecutable y genera el código necesario**
- **Interviene el Linkeditor (Programa). Si hay traducción separada de módulos se enlazan los distintos módulos objeto del programa (módulos, unidades, librerías, procedimientos, funciones, subrutinas, macros, etc.)**
- **Se genera el módulo de carga. Programa objeto completo**
- **Se realiza el proceso de optimización. (Optativo)**
- **El cargador Loader (Programa) lo carga en memoria**

COMPILADORES

○ Optimización:

- **No se hace siempre y no lo hacen todos los compiladores.**
- **Es Optativo**
- **Los optimizadores de código (programas) pueden ser herramientas independientes, o estar incluidas en los compiladores e invocarse por medio de opciones de compilación.**

COMPILADORES

○ Optimización:

- Hay diversas formas y cosas a optimizar:
- elegir entre **velocidad de ejecución y tamaño del código ejecutable.**
- generar código **para un microprocesador** específico dentro de una familia de microprocesadores,
- eliminar la **comprobación de rangos o desbordamientos de pila**
- evaluación para **expresiones booleanas,**
- **eliminación de código muerto** o no utilizado,
- **eliminación de funciones no utilizadas**
- Etc...

COMPILADORES

○ Optimización (ejemplo):

Posibles optimizaciones locales:

Cuando hay **2 saltos seguidos** se puede quedar 1 solo

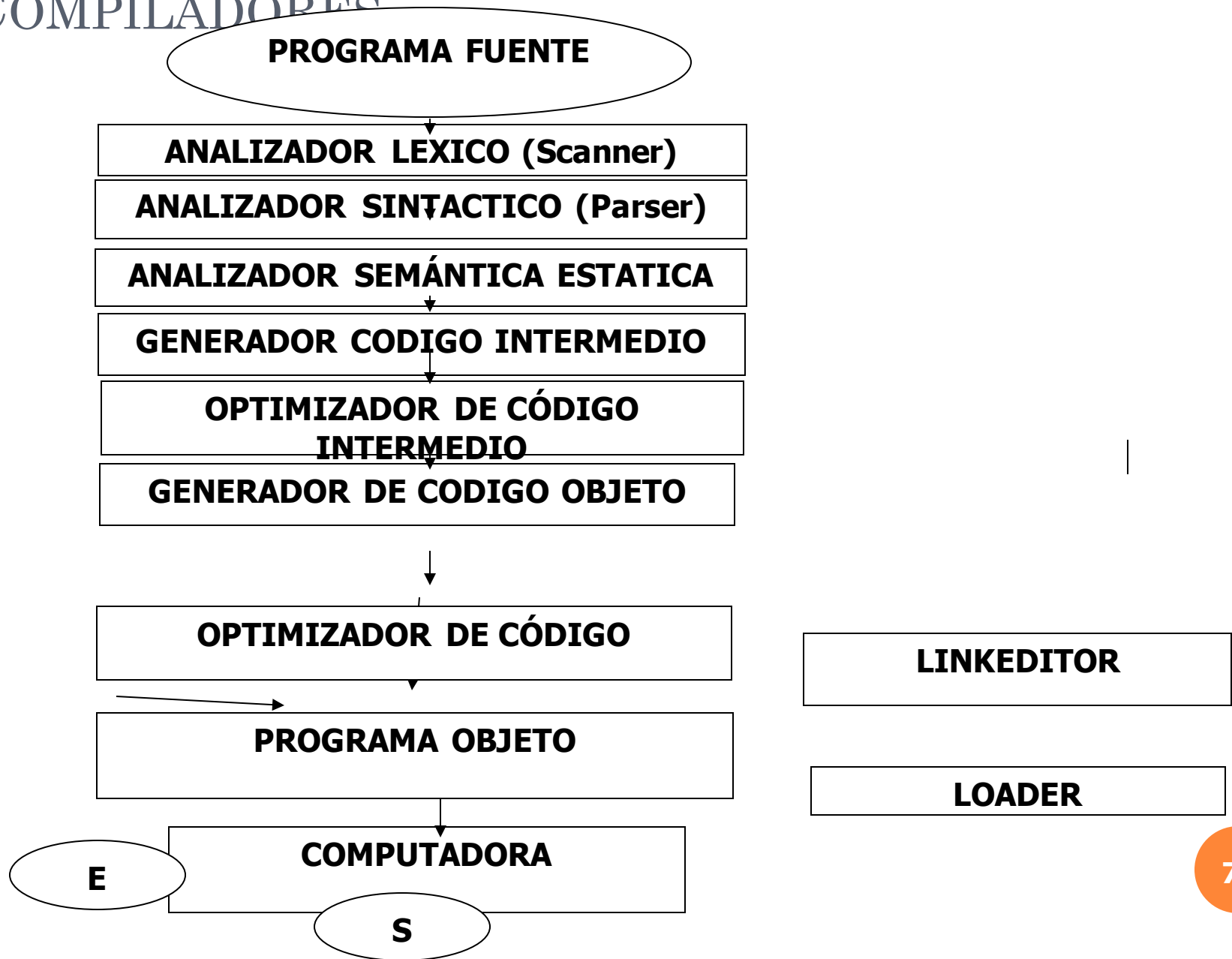
Ejemplo anterior

```
L1: if (a>0) goto L2
      goto L3
L2: t1:=a*4
      t2:=t1-5
      if (b < t2) goto L4
      goto L3
L4: t1:=b*a
      t2:=t1-10
      a:=t2
      goto L1
L3: .....
```

Luego optimización

```
L1: if (a<=0) goto L3
      t1:=a*4
      t2:=t1-5
      if (b >= t2) goto L3
      t1:=b*a
      t2:=t1-10
      a:=t2
      goto L1
L3: .....
```

COMPILADORES



PRÓXIMA CLASE

SEMÁNTICA OPERACIONAL.

- Ligadura. Descriptores. Momentos de ligadura. Estabilidad.
- Variables. Arquitectura Von Newman. Atributos. Momentos y estabilidad. Nombre: características. Alcance: visibilidad, reglas. Tipo: definición, clasificación. L-valor: tiempo de vida, alocação. R-valor: constantes, inicialización. Alias