1. Introducción

- Lenguajes de Programación
- Procesadores de Lenguajes: Compiladores
- Diseño de Lenguajes de Programación y Construcción de Compiladores
- Estructura de un Compilador

Objetivo

Los Lenguajes de Programación (LP) cubren la brecha existente en la especificación de problemas asociada a la comunicación hombre-máquina.

Breve historia del desarrollo de los LP

- **Albores de la informática**
  - Los primeros programas eran cableados o enteramente electromecánicos
  - Posteriormente los programas se codifican como números (Lenguaje Máquina) y se guardan como datos

Ejemplo [Scott, 2009]: Máximo común divisor (MCD) de dos enteros, usando el algoritmo de Euclides, en lenguaje máquina (hexadecimal para x86 de Pentium)

```
55 89 e5 53   83 ec 04 83   e4 f0 e8 31   00 00 00 89   c3 e8 2a 00
00 00 39 c3   74 10 8d b6   00 00 00 00   39 c3 7e 13   29 c3 39 c3
75 f6 89 1c   24 e8 6e 00   00 00 8b ed   fc c9 c3 29   d8 eb eb 90
```


- 1953, IBM lanza el 701 donde la programación se hace en Lenguaje Ensamblador

Ejemplo [Scott, 2009]: MCD en lenguaje ensamblador para x86.

```
pushl  %ebp            cmpl  %eax, %ebx       call   putin
movl   %esp, %ebp      je    C               movl -4( %ebp), %ebx
pushl  %ebx          A: cmpl  %eax, %ebx      leave
subl   $4, %esp        jle   D               ret
andl   $-16, %esp      subl  %eax, %ebx    D: subl   %ebx, %eax
call   getint        B: cmpl  %eax, %ebx      jmp   B
movl   %eax, %ebx      jne   A
call   getint        C: movl  %ebx, ( %esp)
```

- 1957 aparece el primer Lenguaje de Alto Nivel (Fortran) y cambió radicalmente lo que se podía hacer en Informática

Ejemplo: MCD en Fortran.

```
C ** MAXIMO COMUN DIVISOR **
  PROGRAM mcd
  REAL m, n, r, aux
  READ(*,*) m
  READ(*,*) n
  IF (m.LT.n) THEN
      aux = m
      m = n
      n = aux
  ENDIF
  DO WHILE (r.NE.0)
      r = MOD(m,n)
      m = n
      n = r
  ENDDO
  PRINT*, 'es',m
  STOP
  END
```

Características de un Lenguaje Máquina frente a un Lenguaje de Alto Nivel

	Lenguaje Máquina	Lenguaje de Alto Nivel
Objetos a manipular	posiciones y contenidos	objetos complejos
Referencia a los objetos	explícita	implícita
Código y datos	mezclados en memoria	separados en segmentos
Sintaxis	muy simple	compleja

Lenguajes de Programación

- **Después del Fortran** (finales de los años 50)
  - ➢ **Algol** fue la respuesta europea a Fortran: sintaxis moderna, estructura de bloques y declaración explícita
  - ➢ **Lisp** es un lenguaje funcional que implementa λ-cálculos
  - ➢ **Basic** lenguaje de propósito general fácil de usar
  - ➢ **Cobol** se diseño como un lenguaje orientado a los negocios
- **Explosión de lenguajes** (entre los 60 y los 80)
  - ➢ `Algol` derivó en **Pascal** (lenguaje con estructura de bloques de uso académico); **C** (diseñado para el desarrollo de sistemas); y **Simula** (incluye clases y corrutinas)
  - ➢ `Pascal` derivó en **Modula** (incluye concurrencia); y **Ada** (incluye objetos, herencia y facilidades de tiempo real)
  - ➢ `Simula` derivó en **Smalltalk** (lenguaje orientado a objetos)
  - ➢ **ML** lenguaje funcional derivado de `Lisp` y con una sintaxis similar a `Pascal`. `ML` derivó en **Haskell** (lenguaje de funcional más usado)

Lenguajes de Programación

- ➢ **Visual Basic** lenguaje basado en eventos derivado del Basic (por Microsoft).
- ➢ A principios de los 70 aparece **Prolog** (lenguaje de programación lógica más utilizado)
- ➢ A mediados de los 70 surgen diversos lenguajes de *script*: **Perl** (lenguaje de propósito general); y **PHP** (lenguaje orientado al diseño de páginas web)

- **Lenguajes en la actualidad**
  - ➢ **C++** lenguaje orientado a objetos sucesor de `C`, `Smalltalk` y `Ada`
  - ➢ **Java** lenguaje orientado a objetos subconjunto de `C++` para *byte-code*
  - ➢ Proliferación de pequeños lenguajes especializados: **Python** (lenguaje de script orientado a objetos de propósito general derivado de `C++`, `Haskell` y `Perl`); **Ruby** (elegante lenguaje de script orientado a objetos derivado de `Python` y `Smalltalk`); ...

Lenguajes de Programación

¿ Porqué hay tantos ?

- Evolución
- Propósito específico
- Preferencia personal

¿ Porqué un LP tiene éxito ?

- Potencia expresiva
- Baja curva de aprendizaje
- Portabilidad y estandarización
- Buenos compiladores

Lista de lenguajes de programación mas usados (2016), [índice `Tiobe`]

Aug 2016	Aug 2015	Programming Language	Ratings	Change
1	1	`Java`	19.0 %	-0.3 %
2	2	`C`	11.3 %	-3.4 %
3	3	`C++`	5.8 %	-1.9 %
4	4	`C#`	4.9 %	+0.1 %
5	5	`Python`	4.4 %	+0.3 %
6	7	`PHP`	3.2 %	+0.4 %
7	9	`JavaScript`	2.7 %	+0.5 %
8	8	`Visual Basic .NET`	2.5 %	-0.2 %
9	10	`Perl`	2.5 %	+0.4 %
10	12	`Assembly language`	2.4 %	+0.6 %

A good language is one people use

Lenguajes de Programación

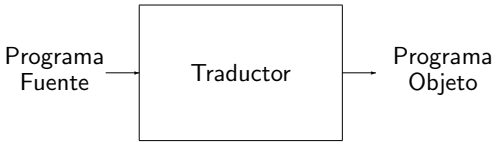
Un buen LP debe ser: preciso, conciso y expresivo
(de Alto Nivel con gran cantidad de posibles abstracciones)

Características esenciales en el diseño de un LP

- **Abstracción** necesaria para construir sistemas complejos
- **Tipos** (inicialmente los LP tenían unos pocos tipos simples)
  - ➢ Tipos permiten al programador expresar abstracciones
  - ➢ Tipos permiten al traductor/compilador detectar errores frecuentes
- **Reutilización** (explotar patrones comunes)
  - ➢ Parametrización de tipos
  - ➢ Herencia
  - ➢ Combinación de ambas (C++, Java)

Modelos de programación y LP asociados

- **Declarativos**
  - ➢ **Funcional**: `Lisp`, `ML`, `Haskell`, ...
  - ➢ **Lógico**: `Prolog`
- **Imperativos**
  - ➢ **Von Neumann**: `Fortran`, `Algol`, `Pascal`, `C`, ...
  - ➢ **Orientado a objetos**: `Simula`, `Smalltalk`, `C++`, `Java`, ...
  - ➢ **Concurente**: `Modula`, `Ada`, ...
  - ➢ **"Script"**: `Perl`, `Python`, `PHP`, ...



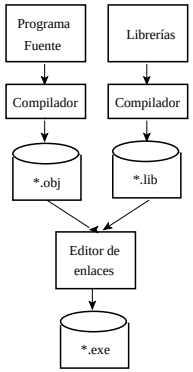
Los Procesadores de Lenguajes (Compiladores) son los traductores de los LP.

Principios fundamentales exigibles a todo Compilador:

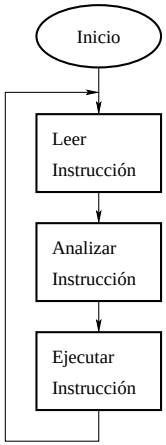
1) El Código Objeto debe ser **correcto** (semánticamente equivalente)
2) El Código Objeto debe ser **eficiente**

Estrategias de implementación de LP

- **Compiladores** El tiempo de traducción es independiente del tiempo de ejecución
- **Intérpretes** El tiempo de traducción está incluido en el de ejecución



Compilador



Interprete

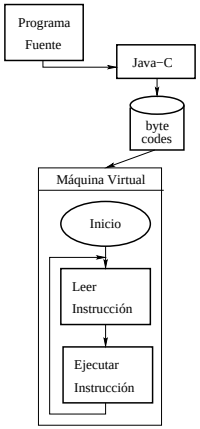
Compiladores

- ✓ El código compilado puede ejecutarse repetidamente y no necesita para ello ni el código fuente ni que esté presente el propio compilador
- ✓ Tratamiento de errores eficiente en tiempo de traducción
- ✓ Pueden abordarse optimizaciones de gran calado
- ✗ Las comprobaciones estáticas tiene una capacidad limitada
- ✗ No se pueden modificar los programas sobre la marcha

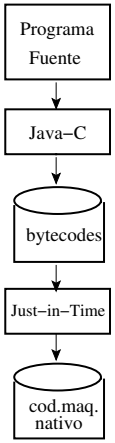
Intérpretes

- ✗ Traduce las instrucciones cada vez que estas se ejecutan, se necesita tanto el código fuente como el propio intérprete para la ejecución del programa
- ✗ La mayor parte de la detección de los errores se produce en tiempo de ejecución
- ✗ No se pueden desarrollar optimizaciones de código importantes
- ✓ Facilidad para depurar programas, se tiene un mayor control sobre el comportamiento del programa
- ✓ El código no depende de la plataforma
- ✓ Se puede modificar dinámicamente el programa

Procesadores de Lenguajes: Compiladores

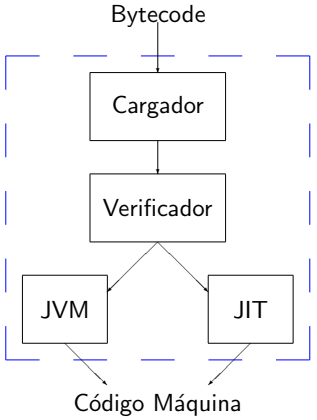


Interprete de `Bytecodes`



Compilador `Just-in-Time`

Procesadores de Lenguajes: Compiladores



Plataforma `Hot Spot`

Procesadores de Lenguajes: Compiladores

Otros tipos de traductores:

- Conversor fuente-fuente
- Compilador cruzado
- Compilador incremental
- Decompiladores

Conceptos relacionados con la compilación

- Editores de enlaces y Cargadores
- Preprocesadores
- Editores especializados
- Entornos de desarrollo

Otras aplicaciones de las técnicas de compilación:

- Formateadores de texto
- Intérpretes de comandos
- Intérpretes de consultas a bases de datos
- Módulos de análisis en aplicaciones que aceptan ficheros de intercambio
- Lenguaje natural y reconocimiento sintáctico de formas

Diseño de LP y construcción de Compiladores

Especificación de los LP

Especificación Léxica — Expresiones Regulares

Especificación Sintáctica — Gramáticas Incontextuales

Especificación Semántica

Para comprobaciones semánticas estáticas — Gramáticas de Atributos

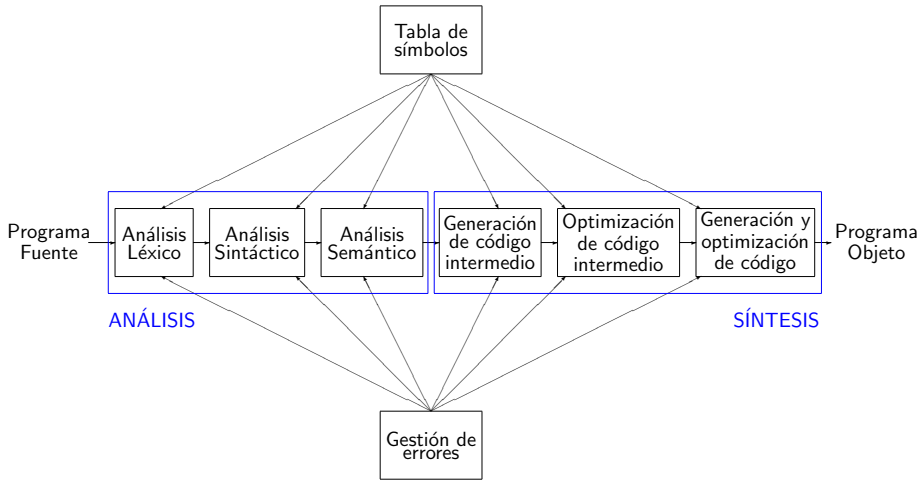
Para definir la dinámica de la ejecución

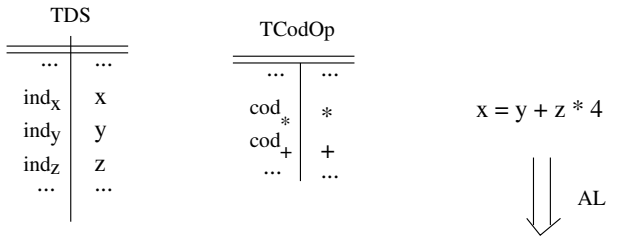
- Semántica axiomática
- Semántica denotacional
- Semántica operacional

- Semántica axiomática (Floyd, Hoare)
  - El significado se describe mediante sentencias lógicas
  - Énfasis en *demostrar* la corrección del programa
  - Útil para los usuarios del lenguaje
- Semántica denotacional (Strachey, Scott)
  - El significado viene dado por un objeto matemático (una función)
  - Énfasis en *describir* las transformaciones efectivas
  - Útil para los diseñadores del lenguaje
- Semántica operacional (Plotkin)
  - El significado se define mediante una máquina abstracta (con estados)
  - Énfasis en *explicar* como se ejecuta en una máquina abstracta
  - Útil para los implementadores del lenguaje
  - La implantación de un lenguaje es *correcta* sólo si las semánticas denotacional y operacional concuerdan

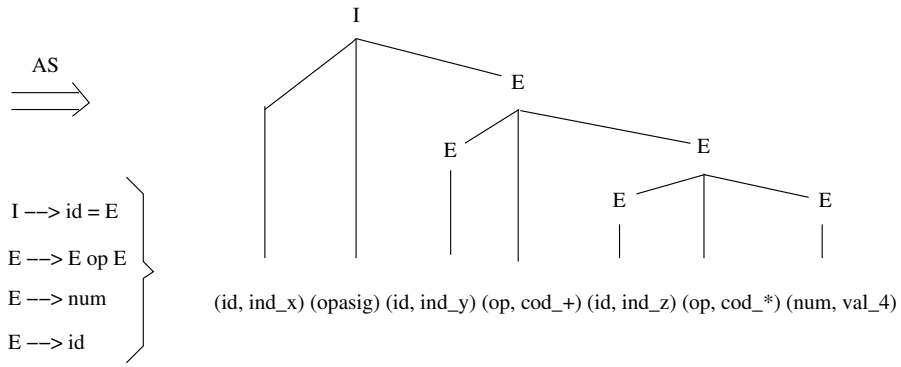
Especificación de los LP & diseño de sus traductores

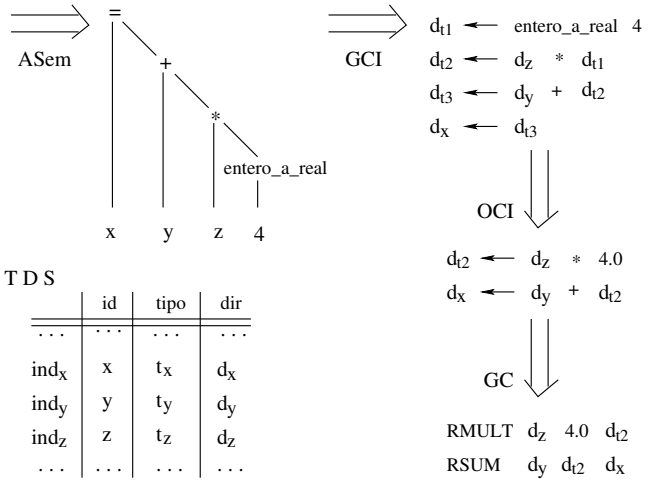
Lenguajes de programación	Traductores
Especificación léxica expresiones regulares	Análisis léxico autómatas finitos
Especificación sintáctica gramáticas incontextuales	Análisis sintáctico autómatas a pila
Especificación semántica estática gramáticas de atributos	Análisis semántico estático esquemas de traducción





(id, ind_x) (opasig) (id, ind_y) (op, cod_+) (id, ind_z) (op, cod_*) (num, val_4)


AS
I
E
E
E
E
E
I —> id = E
E —> E op E
E —> num
E —> id
(id, ind_x) (opasig) (id, ind_y) (op, cod_+) (id, ind_z) (op, cod_*) (num, val_4)


ASem
=
+
*
entero_a_real
x
y
z
4
T D S
id
tipo
dir
ind_x x t_x d_x
ind_y y t_y d_y
ind_z z t_z d_z
GCI
d_t1 ← entero_a_real 4
d_t2 ← d_z * d_t1
d_t3 ← d_y + d_t2
d_x ← d_t3
OCI
d_t2 ← d_z * 4.0
d_x ← d_y + d_t2
GC
RMULT d_z 4.0 d_t2
RSUM d_y d_t2 d_x