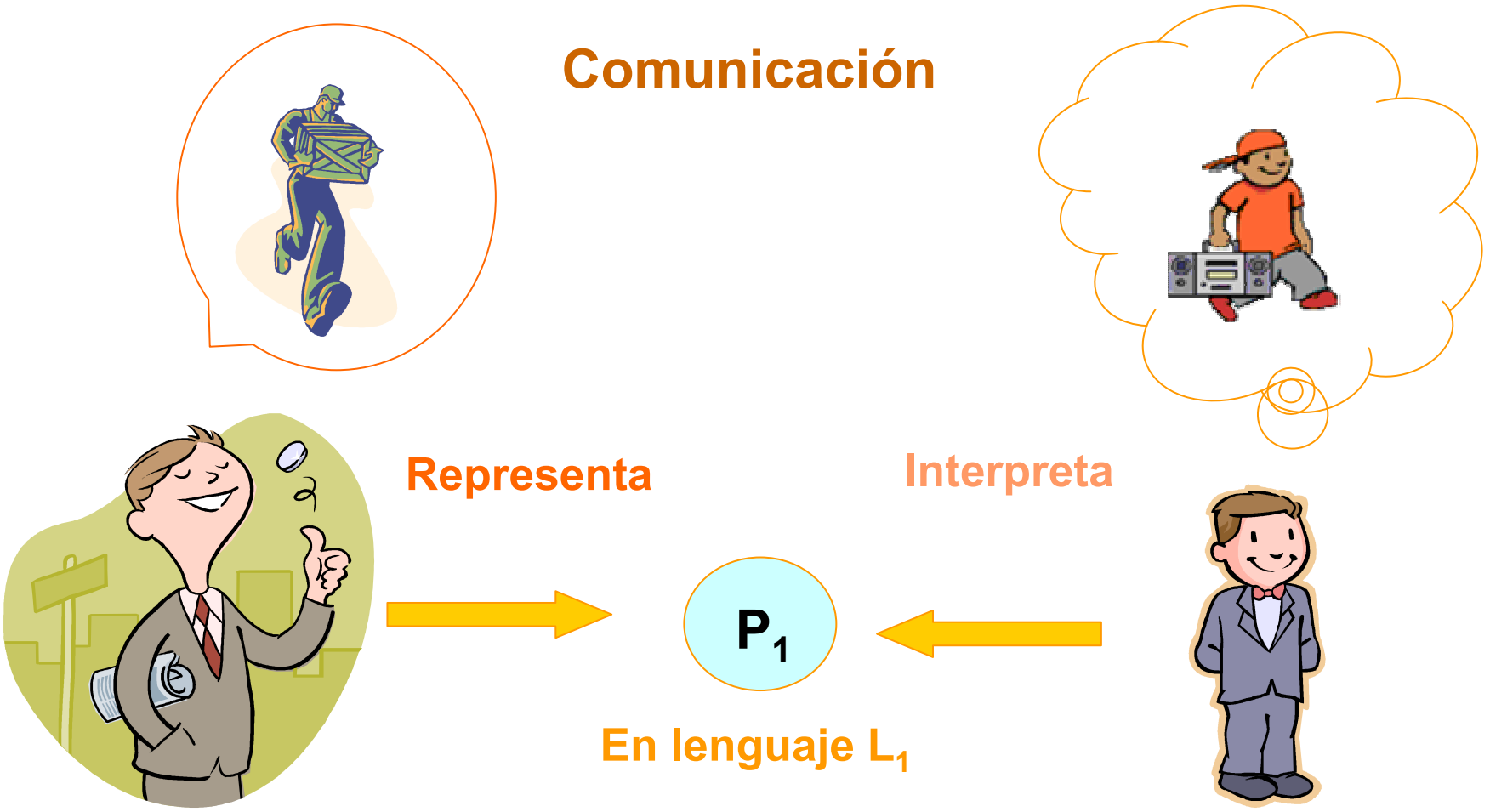
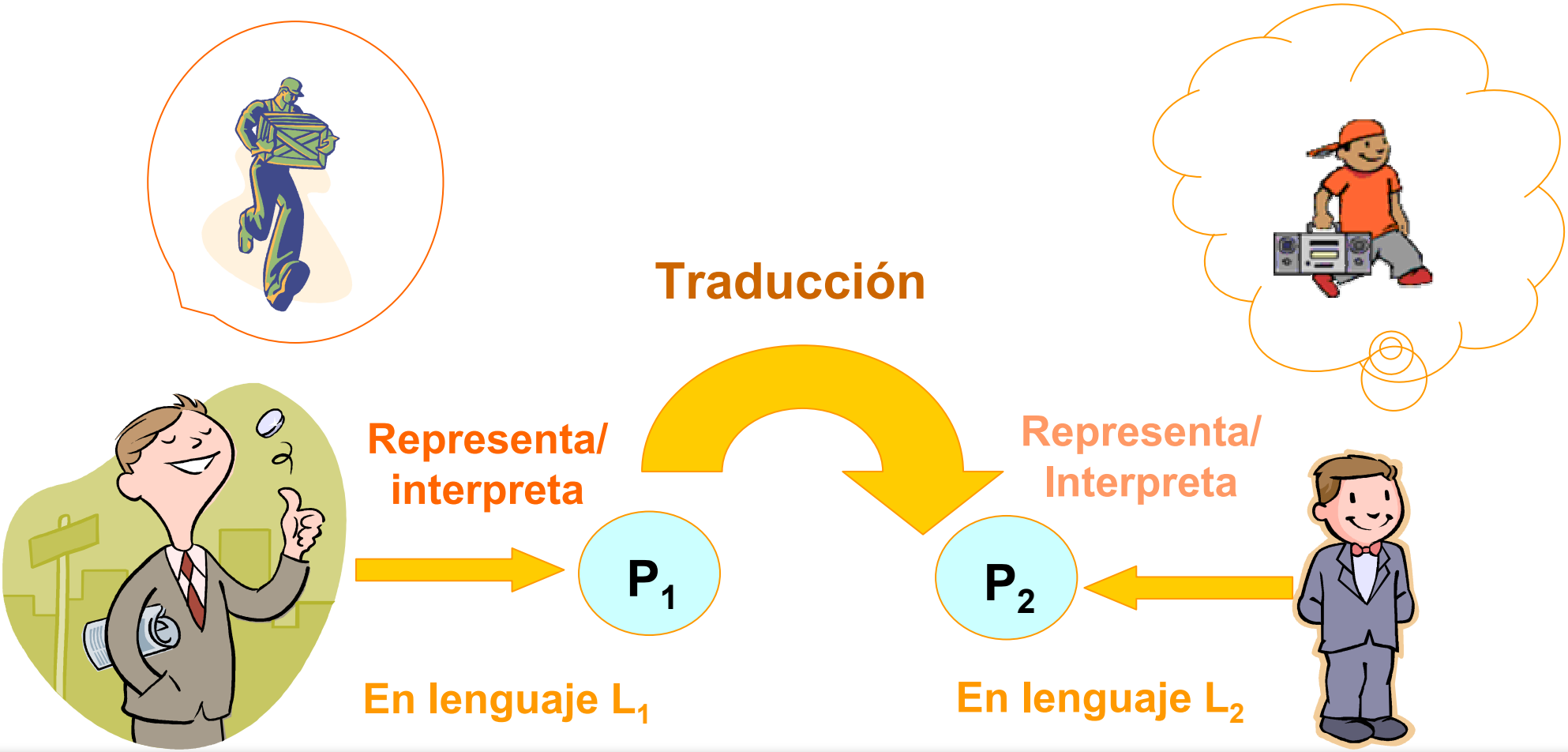
Tema 1 | Procesadores de Lenguajes

Contenidos

1.1.1 Concepto de traducción

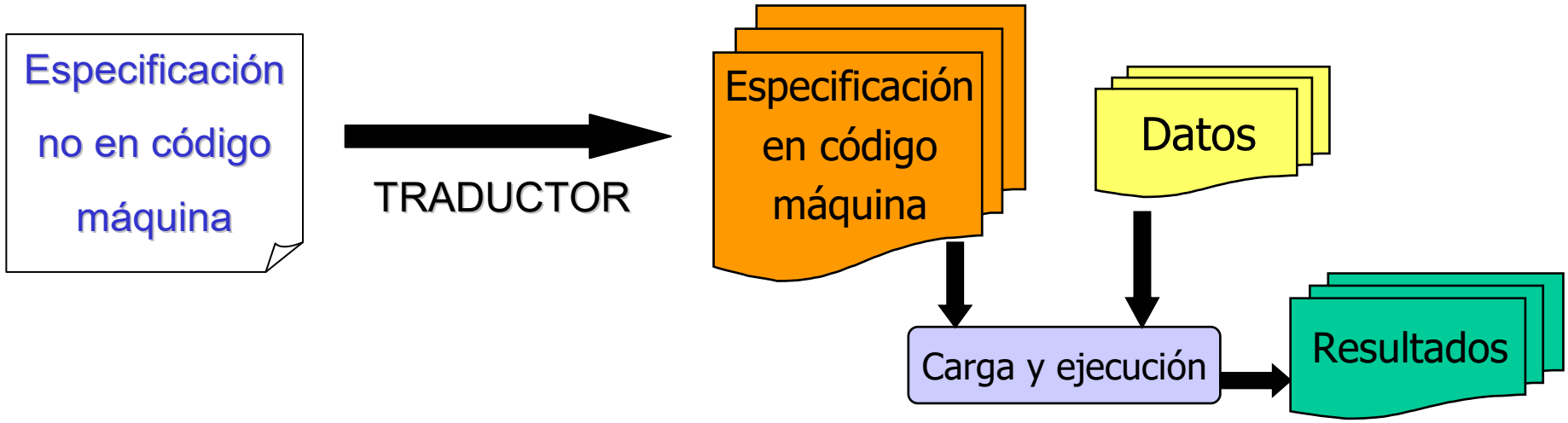


1.1.1 Concepto de traducción



1.1.1 Concepto de traducción

Cuando el lenguaje de especificación de un programa es diferente al lenguaje máquina es necesario traducirlo a lenguaje máquina para poder ejecutarlo.



Todo lenguaje de programación (no máquina) define una **MÁQUINA VIRTUAL**

1.1.1 Concepto de traducción

- La traducción relaciona a dos o más **modelos semánticos (culturas)**
- Un modelo semántico está formado por:

Lenguaje

Dominio semántico

Interpretación

1.1.1 Concepto de traducción. Lenguaje

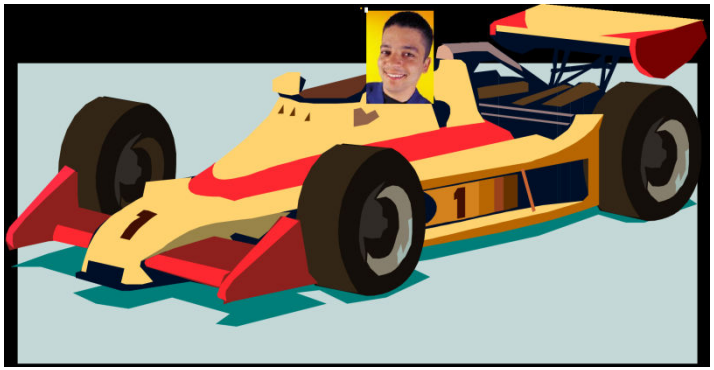
- Conjunto de construcciones simbólicas formadas por secuencias de símbolos de un alfabeto
- Se utiliza para representar los valores semánticos de un dominio semántico
- Un lenguaje puede definirse:
  - Por Extensión
  - Por Comprensión → GRAMÁTICA

1.1.1 Concepto de traducción. Dominio semántico

Valores semánticos
- **Elementales**
- **Estructurados**

coche

Juan



Juan conduce el coche

1.1.1 Concepto de traducción: Interpretación

Sea *D* un dominio semántico y *L(G)* el lenguaje definido por la gramática *G*. Se define la *interpretación* como una aplicación *I*:

$$I : L(G) \longrightarrow D$$

Se denota el modelo semántico (o modelo de la interpretación) como:

$$M = (L(G), D, I)$$

1.1.2 Requisitos para construir un traductor

- Dados dos lenguajes $L_1(G_1)$ y $L_2(G_2)$

Se necesitan dos modelos semánticos:

$$M_1 = (L_1(G_1), D, I_1) \text{ y } M_2 = (L_2(G_2), D, I_2)$$

Tal que se debe cumplir:

$$D_1 \subseteq D_2 \text{ y}$$

$$\forall \alpha \in L_1(G_1),\ \exists \beta \in L_2(G_2) \ / \ I_1(\alpha) = I_2(\beta)$$

1.1.3 Esquema de Traducción

- Dados los modelos semánticos:

$$M_1 = (L_1(G_1), D_1, I_1) \; y \; M_2 = (L_2(G_2), D_2, I_2)$$

Se define un *Esquema de Traducción* como una función *T* :

$$T : L_1(G_1) \longrightarrow L_2(G_2)$$

verificándose que:

$$\forall \alpha \in L_1(G_1), \exists \beta \in L_2(G_2) \; con \; T(\alpha) = \beta \; y \; I_1(\alpha) = I_2(\beta)$$

Tema 1 | Procesadores de Lenguajes

Contenidos

1.2 Aplicaciones de los traductores

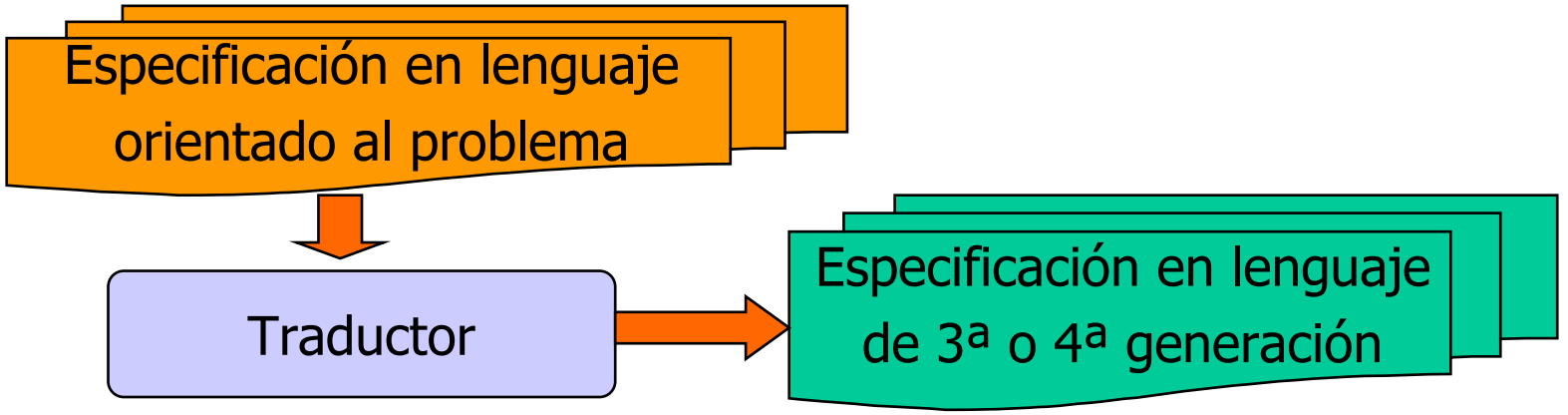
- Lenguajes de programación
- Lenguajes orientados al problema
- Macroensambladores
- Manejadores de bases de datos

1.2 Aplicaciones de los traductores

- Lenguajes de programación
  - Traducir a código máquina programas escritos en C, Pascal, etc.
  - El dominio semántico (evaluación y control) de los lenguajes de programación coincide con el dominio semántico del lenguaje máquina.

1.2 Aplicaciones de los traductores: Lenguajes orientados al problema

- El problema se puede expresar en términos de valores semánticos elementales y estructurados.
- Podemos definir un lenguaje orientado al problema que permita formular el problema de forma más cercana a la cultura del usuario.
- Construir su traductor (compilador o intérprete).



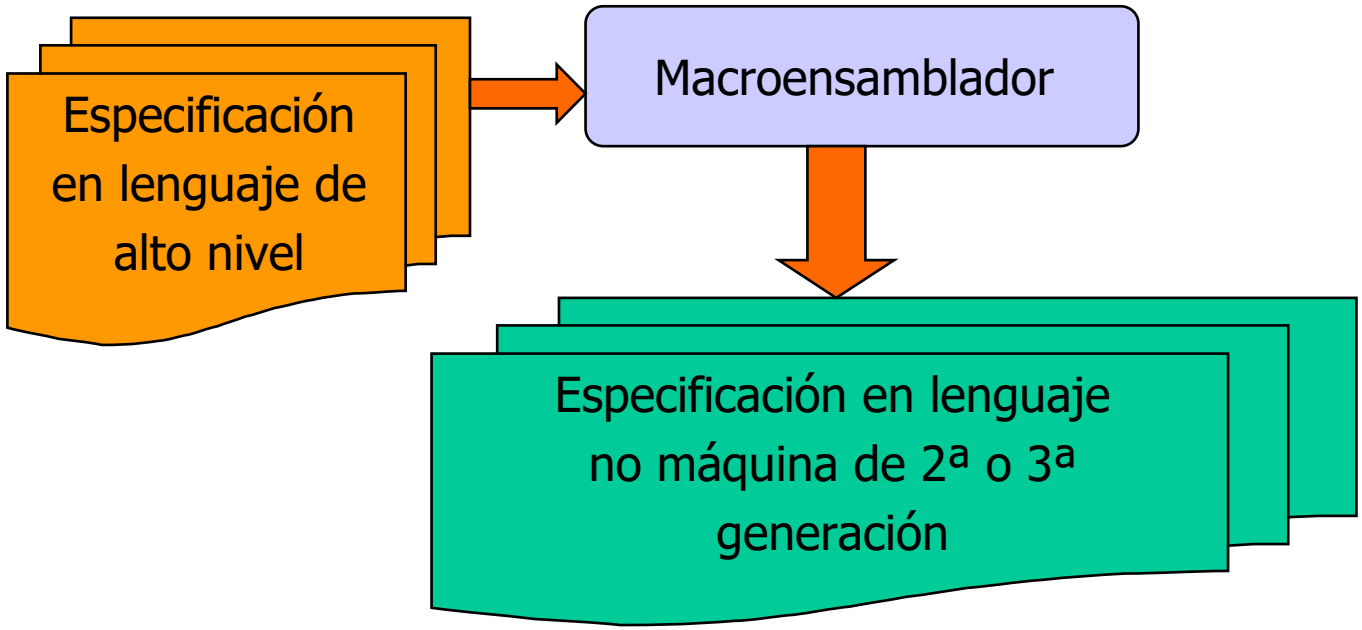
1.2 Aplicaciones de los traductores: Lenguajes orientados al problema (ejemplo)

- Imaginemos la gestión de un almacén, donde se dispone de *productos*, *clientes*, *proveedores*, así como de las acciones de *compra*, *venta*, *baja* y *pedido*.
- Se podría diseñar un lenguaje en donde se expresaran las acciones de la forma:

  - `venta cantidad_1 articulo_1 a cliente_1`
  - `si cantidad_2 < 20 entonces pedido articulo_1 a proveedor_1`
  - `baja cantidad_1 existencia actual cantidad_2`

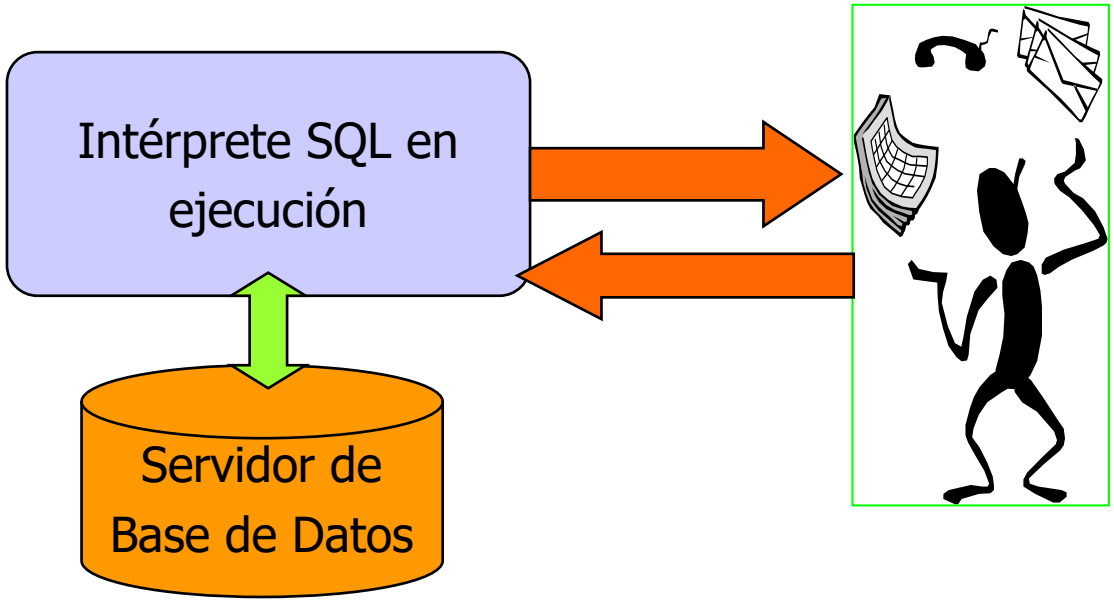
- Este lenguaje nos permitiría poder expresar nuestro problema a nivel de diseño, permitiendo realizar la fase de codificación de forma automática, mediante un traductor desde el lenguaje diseñado a un lenguaje de alto o bajo nivel.

1.2 Aplicaciones de los traductores: Macroensambladores



1.2 Aplicaciones de los traductores: Manejadores de bases de datos

- Las instrucciones se procesan como las especificaciones para macroensambladores.
- Generan llamadas a funciones de acceso al DBMS como interpretación de las instrucciones de alto nivel.



Tema 1 | Procesadores de Lenguajes

Contenidos

1.3 Visión general de un traductor

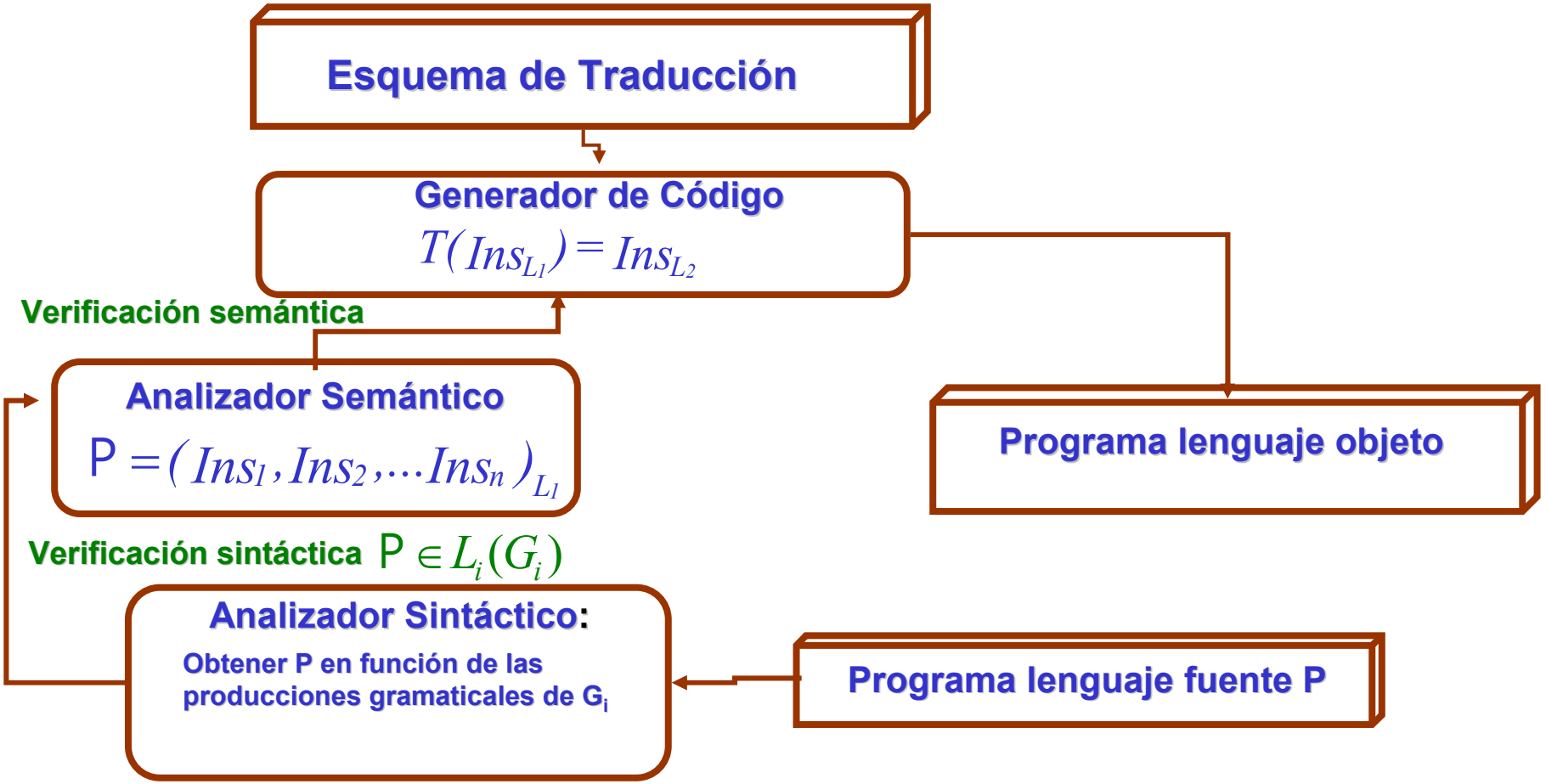
Dados los modelos semánticos $\begin{cases} M_1 = (L_1(G_1), D_1, I_1) \\ M_2 = (L_2(G_2), D_2, I_2) \end{cases}$ y

y su esquema de traducción $T : L_1(G_1) \longrightarrow L_2(G_2)$

Para traducir un programa $P \in L_1(G_1)$

¿Qué tareas y en qué orden se deben realizar?

1.3 Visión general de un traductor



1.3 Visión general de un traductor

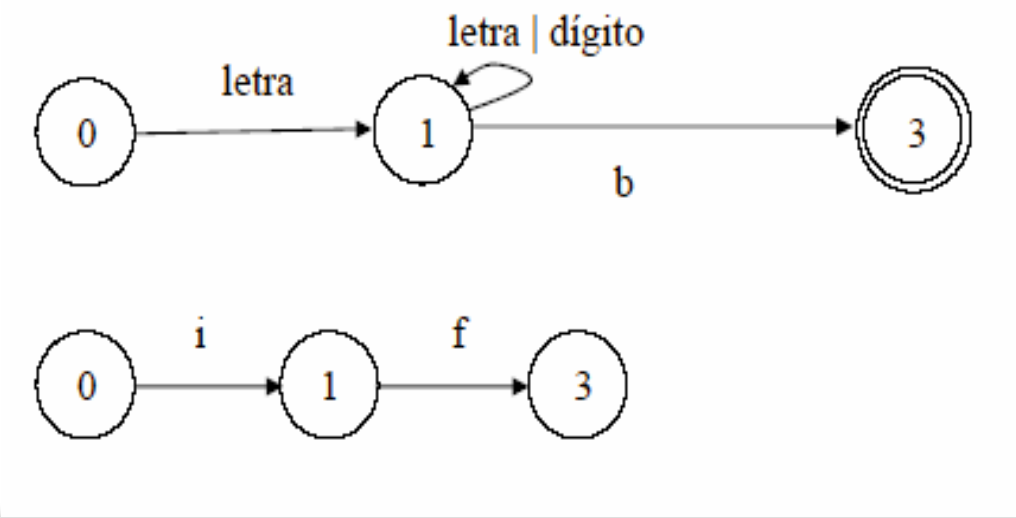
- Fases
  - Análisis
    - Verificación sintáctica
      - Análisis léxico
      - Análisis sintáctico
    - Verificación semántica
      - Análisis semántico
  - Síntesis
    - Generación de código
    - Optimización de código

1.3.2 Fase de análisis. Verificación sintáctica

- La verificación sintáctica se puede simplificar teniendo en cuenta lo siguiente:
  - Elegir lenguajes que estén definidos por **gramáticas regulares** (tipo 3) o **gramáticas independientes del contexto** (tipo 2).
  - Diseñar lenguajes formados como concatenación de **palabras** y las palabras formadas como concatenación de símbolos de un alfabeto.
  - Si las frases obedecen a una estructura sintáctica, la verificación sintáctica se puede descomponer en:
    - **Análisis léxico:** Se identifican las palabras con la misma misión sintáctica.
    - **Análisis sintáctico:** Se verifica la sintaxis de la secuencia de palabras. Se pueden aplicar **técnicas predictivas** (sin vuelta atrás).

1.3.2 Fase de análisis. Verificación Sintáctica: Análisis léxico

- **Componente léxico (símbolo o token):** Conjunto de palabras que hacen la misma misión sintáctica. Ejemplo verbo, sujeto, …. Los tokens son definidos por expresiones regulares (sublenguaje).
- **Lexema:** Cada palabra concreta del texto fuente asociada a un token.
- **Patrón:** Regla mediante la cual una secuencia de caracteres del texto fuente es asociada a un token (regla de formación de un token). Cada patrón es reconocido por un Autómata Finito Determinista (AFD).



1.3.2 Fase de análisis. Verificación sintáctica: Análisis léxico

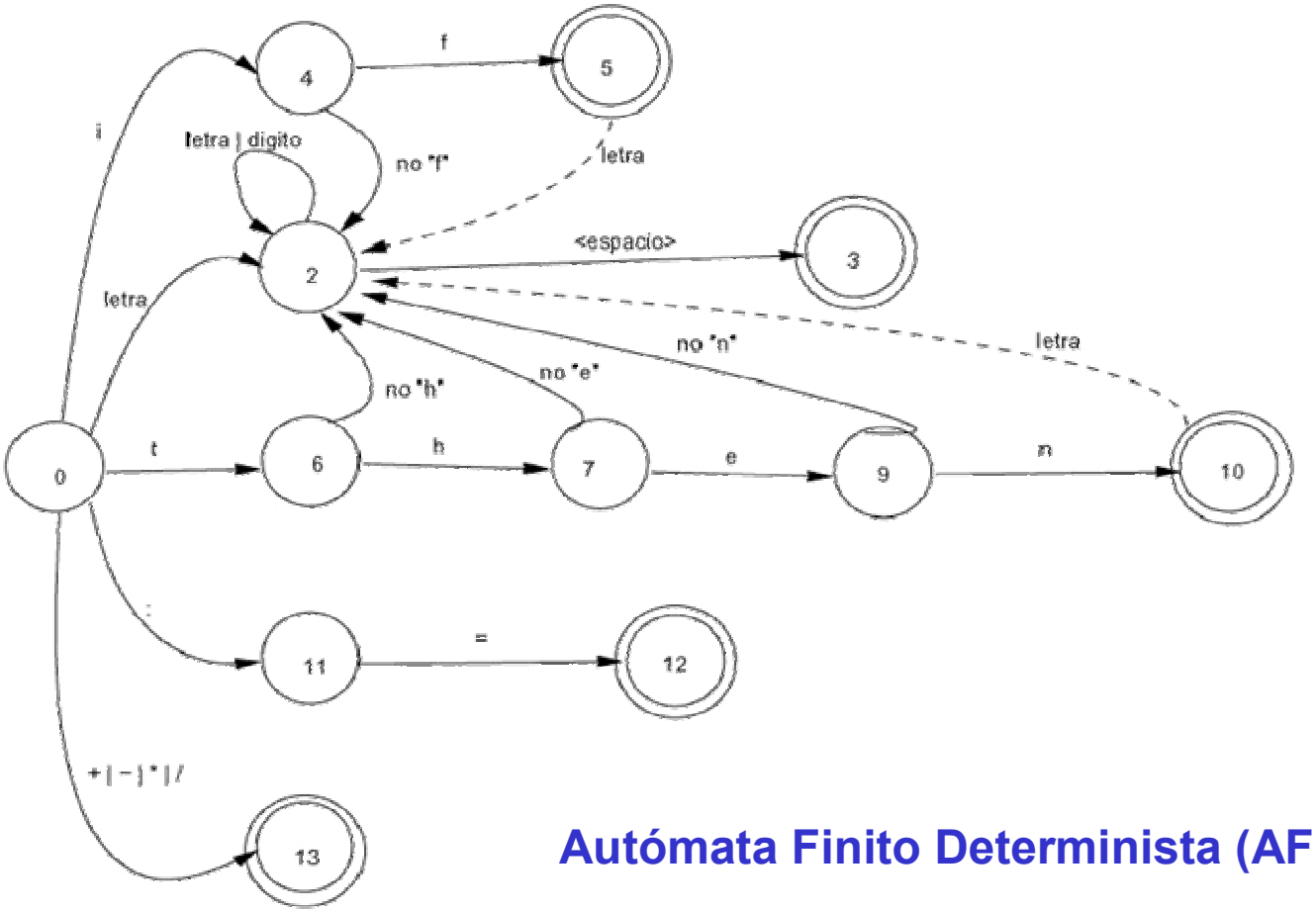
$$
\begin{aligned}
S &\rightarrow A \mid C \\
A &\rightarrow id := E \\
C &\rightarrow if\ E\ then\ S \\
E &\rightarrow EOE \mid (E) \mid id \\
O &\rightarrow + \mid - \mid * \mid / \\
id &\rightarrow letra \mid id\ digito \mid id\ letra \\
letra &\rightarrow a|b|\cdots|z \\
digito &\rightarrow 0|1|\cdots|9
\end{aligned}
$$

Token	Patrón / Expresión regular	Código
ID	`letra(letra|digito)*`	257
ASIGN	`":="`	258
IF	`"if"`	259
THEN	`"then"`	260
PARIZQ	`"("`	261
PARDER	`")"`	262
OPEBIN	`"+"|"-"|"*"|"/"`	263

Ejemplo:

contador := su39 + suma * (a30r / cantidad)

1.3.2 Fase de análisis. Verificación sintáctica: Análisis léxico



Autómata Finito Determinista (AFD) con anticipación

1.3.2 Fase de análisis. Verificación sintáctica: Análisis léxico

- Desde el análisis sintáctico, se obvia el lexema de cada componente léxico, sólo nos interesa la presencia de un determinado componente léxico. Pero desde la perspectiva del análisis semántico y generación de código, se necesita conocer el lexema concreto.

- ¿Cómo se soluciona este problema?. Introduciendo los lexemas en una tabla de símbolos, y conservando para cada componente léxico un atributo que indica el lexema concreto que representa.

- Herramienta para construir el analizador de léxico: LEX

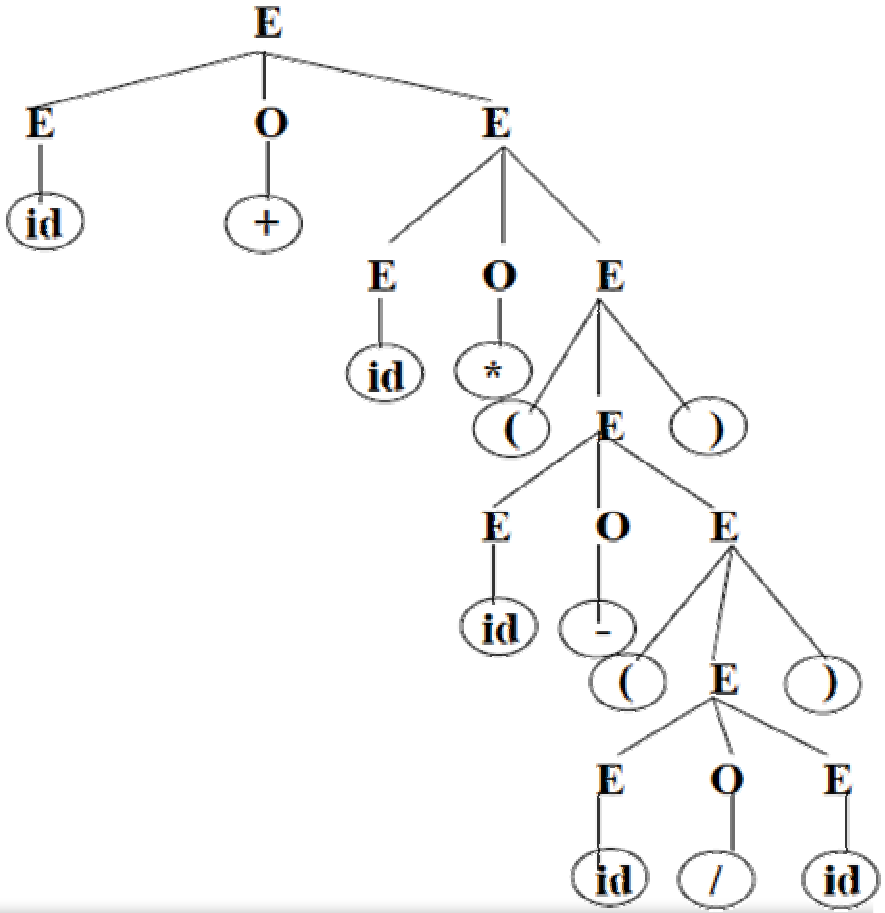
1.3.2 Fase de análisis. Verificación sintáctica: Análisis sintáctico

- La complejidad del análisis sintáctico depende del tipo de gramática que define el lenguaje.
- Una gramática G se define mediante una cuádrupla, **G = (N, T, P, S),** donde:
  - **N** representa el conjunto de símbolos no terminales
  - **T** es el conjunto de símbolos terminales
  - **P** conjunto de producciones (o reglas) gramaticales
  - **S** símbolo (no terminal) inicial de la gramática

1.3.2 Fase de análisis. Verificación sintáctica: Análisis sintáctico

$$
\begin{aligned}
P &= \{ \quad E \rightarrow EOE \\
&\qquad\quad E \rightarrow (E) \\
&\qquad\quad E \rightarrow id \\
&\qquad\quad O \rightarrow + \mid - \mid * \mid / \ \}
\end{aligned}
$$

$$
\begin{aligned}
N &= \{E, O\} \\
T &= \{id, (,), +, -, *, /\} \\
S &= E
\end{aligned}
$$



$$id + id * (id - (id / id))$$

1.3.2 Fase de análisis. Verificación sintáctica: Análisis sintáctico
Clasificación de las Gramáticas (según Chomsky)

Tipo 3	**Gramáticas regulares** (Secuencias de símbolos: Palabras) $P = \{A \to aB\ ;\ A \to a\ /\ a \in T,\ B \in (V_N \cup V_T)^*\}$ Son reconocidas por un *autómata finito*
Tipo 2	**Gramáticas independientes del contexto** (Secuencias de palabras) $P = \{A \to \alpha\ /\ A \in V_N\ ;\ \alpha \subset (N \cup T)^*\}$ Son reconocidas por un *autómata con pila (push-down)*
Tipo 1	**Gramáticas sensibles al contexto** $P = \{\alpha A \beta \to \alpha\gamma\beta\ /\ A \in V_N\ ;\ \gamma \subset (V_N \cup V_T)^+\ ;\ \alpha, \beta \subset (V_N \cup V_T)^*\}$ Son reconocidas por un *autómata lineal limitado*
Tipo 0	**Gramáticas con estructura de frase** $P = \{\alpha \to \beta\ /\ \alpha \subset (V_N \cup V_T)^+\ ;\ \beta \subset (V_N \cup V_T)^*\}$ Son reconocidas por una *máquina de Turing*

1.3.2 Fase de análisis. Verificación sintáctica: Análisis sintáctico

- Para asegurar que una cadena de símbolos pertenece a un lenguaje, es necesario aplicar todas las combinaciones posibles sobre las producciones de la gramática y comprobar si puede generarse. Esto no siempre es posible.
- Supongamos la entrada:

$$id + id * (id - (id / id))$$

- y supongamos la gramática:

$$E \rightarrow EOE,\ E \rightarrow id,\ O \rightarrow + \mid -$$

- La aplicación de producciones nunca terminaría.
- ¿Cómo solucionamos este problema? Análisis sintáctico predictivo.

1.3.2 Fase de análisis. Verificación sintáctica: Análisis sintáctico

Abstracción léxica: Simplificar la gramática original en otra gramática que denominaremos gramática abstracta.

$$
\begin{array}{rcl}
S & \rightarrow & A \mid C \\
A & \rightarrow & id := E \\
C & \rightarrow & if\ E\ then\ S \\
E & \rightarrow & EOE \mid (E) \mid id \\
O & \rightarrow & + \mid - \mid * \mid / \\
id & \rightarrow & letra \mid id\ digito \mid id\ letra \\
letra & \rightarrow & a|b|\cdots|z \\
digito & \rightarrow & 0|1|\cdots|9
\end{array}
\quad \Rightarrow \quad
\begin{array}{rcl}
S & \rightarrow & A \mid C \\
A & \rightarrow & \textbf{ID ASIGN } E \\
C & \rightarrow & \textbf{IF } E \textbf{ THEN } S \\
E & \rightarrow & E \textbf{ OPBIN } E \\
 & \mid & \textbf{PARIZQ } E \textbf{ PARDER} \\
 & \mid & \textbf{ID}
\end{array}
$$

Estrategia de análisis { **Descendente** / **Ascendente**

1.3.2 Fase de análisis. Verificación sintáctica: Análisis sintáctico predictivo

- Ante un símbolo de entrada, permite decidir qué hacer para proseguir el análisis y obtener una cadena del lenguaje, sin tener que generar todas las posibles cadenas del lenguaje.
- Es válido si la gramática no es ambigua.
- Existen técnicas de análisis predictivas dependiendo de la estrategia de análisis.
  - Descendente: LL(1)
  - Ascendente:
    - Precedencia
    - LR
    - SLR
    - LALR
- **Herramienta para realizar el análisis sintáctico: YACC**

1.3.2 Fase de análisis. Verificación semántica: Análisis semántico

- Desde el árbol sintáctico (secuencia de producciones aplicadas en el análisis) podemos identificar la secuencias de componentes léxicos con significado conjunto.

- Ya que partimos de una secuencia de **componentes léxicos** correcta sintácticamente, podemos analizar el **valor semántico estructurado** asociado con cada subsecuencia de componentes léxicos junto con su argumentos.

1.3.2 Fase de análisis. Verificación semántica: Análisis semántico

- En el ejemplo anterior, tenemos dos símbolos no terminales de la gramática que identifica secuencias de entrada con significado conjunto:

```
C → if E then S ----------→  C → IF E THEN S
A → id := E     ----------→  A → ID ASIGN E
```

- Al aplicar la producción **A → ID ASIGN E**, debemos comprobar que el lexema asociado con **ID** debe ser del mismo tipo que el de la expresión **E**. Esto nos facilita completar la información acerca de los lexemas. Nos permite completar la información de la tabla de símbolos.

- Verificaciones semánticas.

1.3.3 Fase de síntesis: Generación de código

- Consiste en aplicar de forma particularizada el **Esquema de Traducción** para cada secuencia de lexemas con sentido propio.

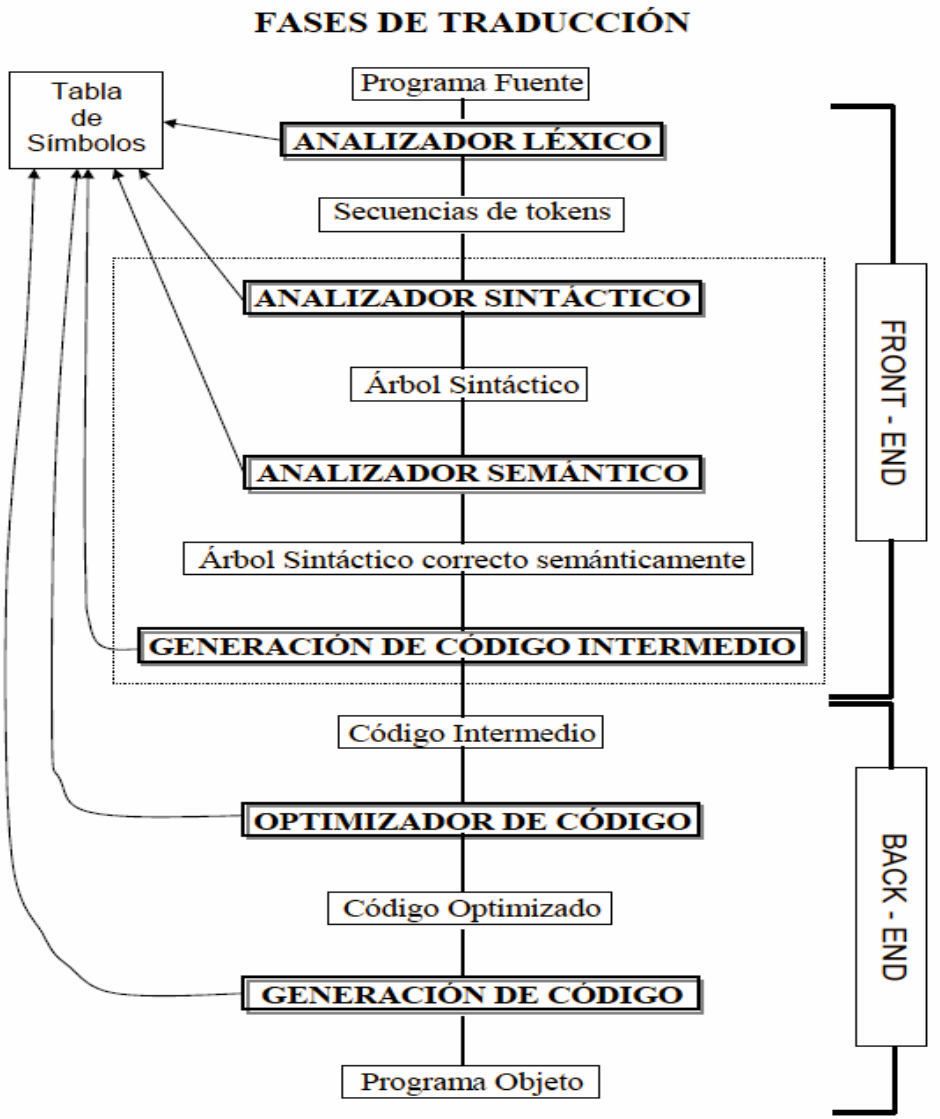
- Un aspecto importante en los traductores de lenguajes de programación consiste en que deben generar códigos de una eficiencia comparable al generado directamente en lenguaje máquina.

- Esta cualidad deseable de los traductores ha dado lugar al desarrollo de técnicas de optimización de código.

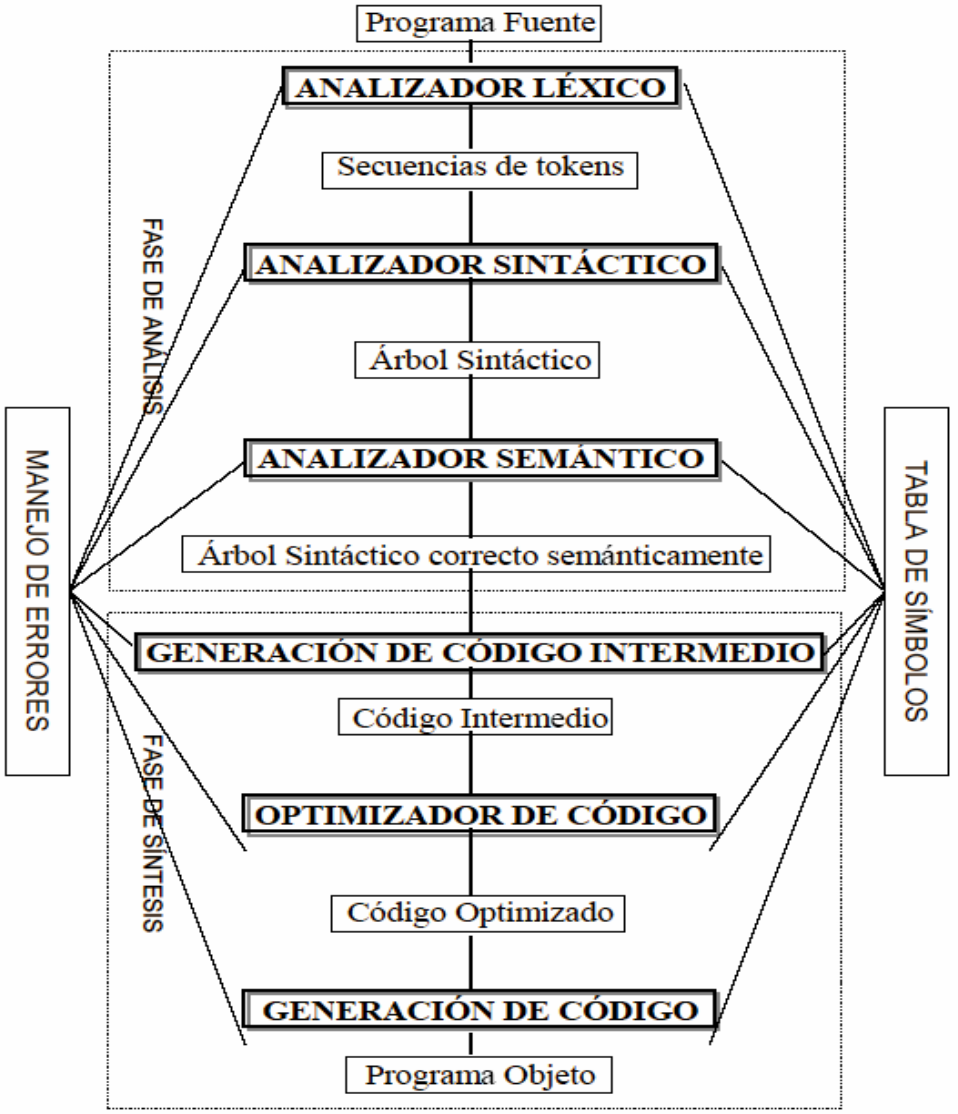
1.1.3 Fase de síntesis: Optimización de código

- Toda secuencia de programa obtenido en código máquina u objeto se caracteriza por la presencia de:
  - Instrucciones de asignación
  - Instrucciones de evaluación
  - Instrucciones de control
- En ocasiones es posible reorganizar el código para reducir el número de instrucciones

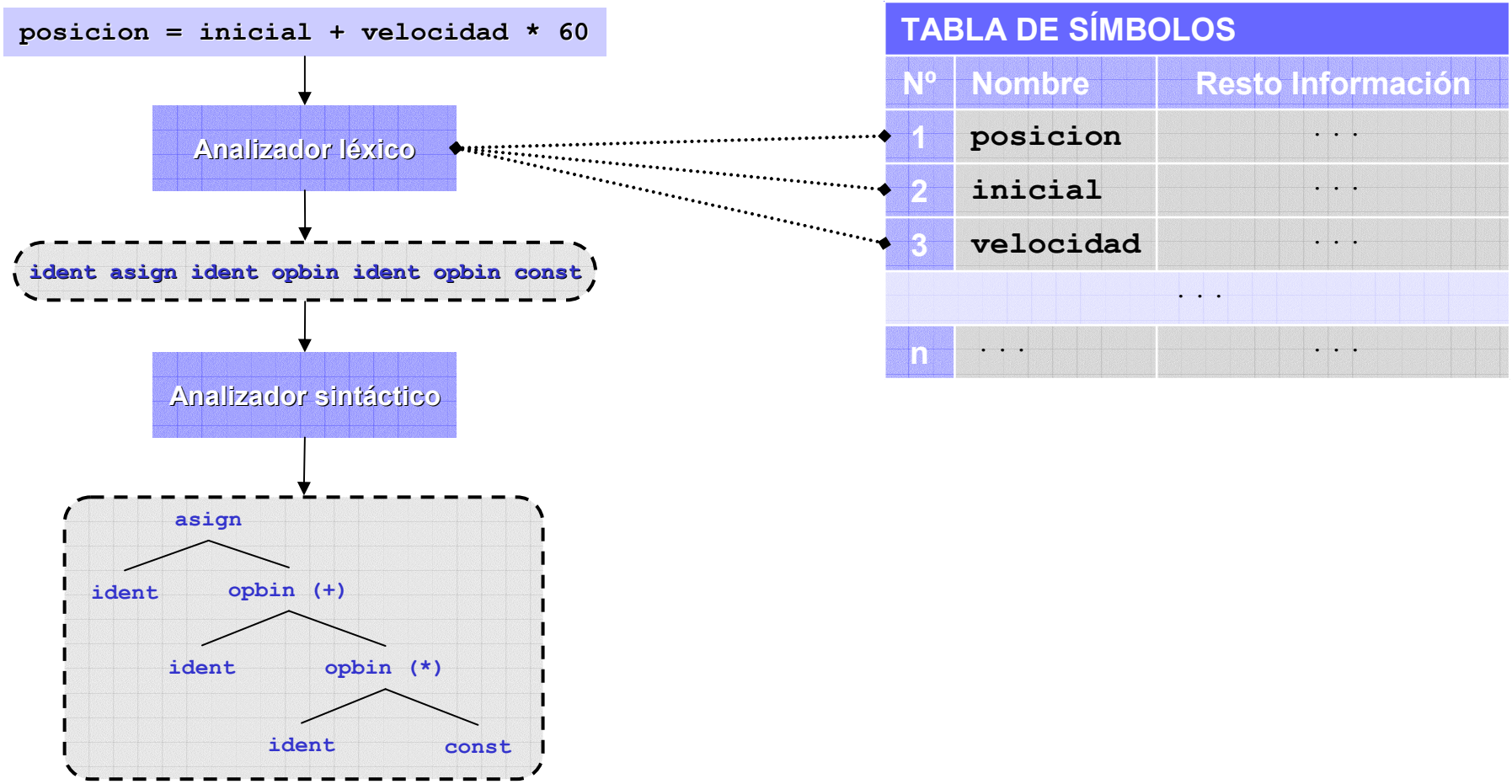
1.1.4 Modelo general de un traductor



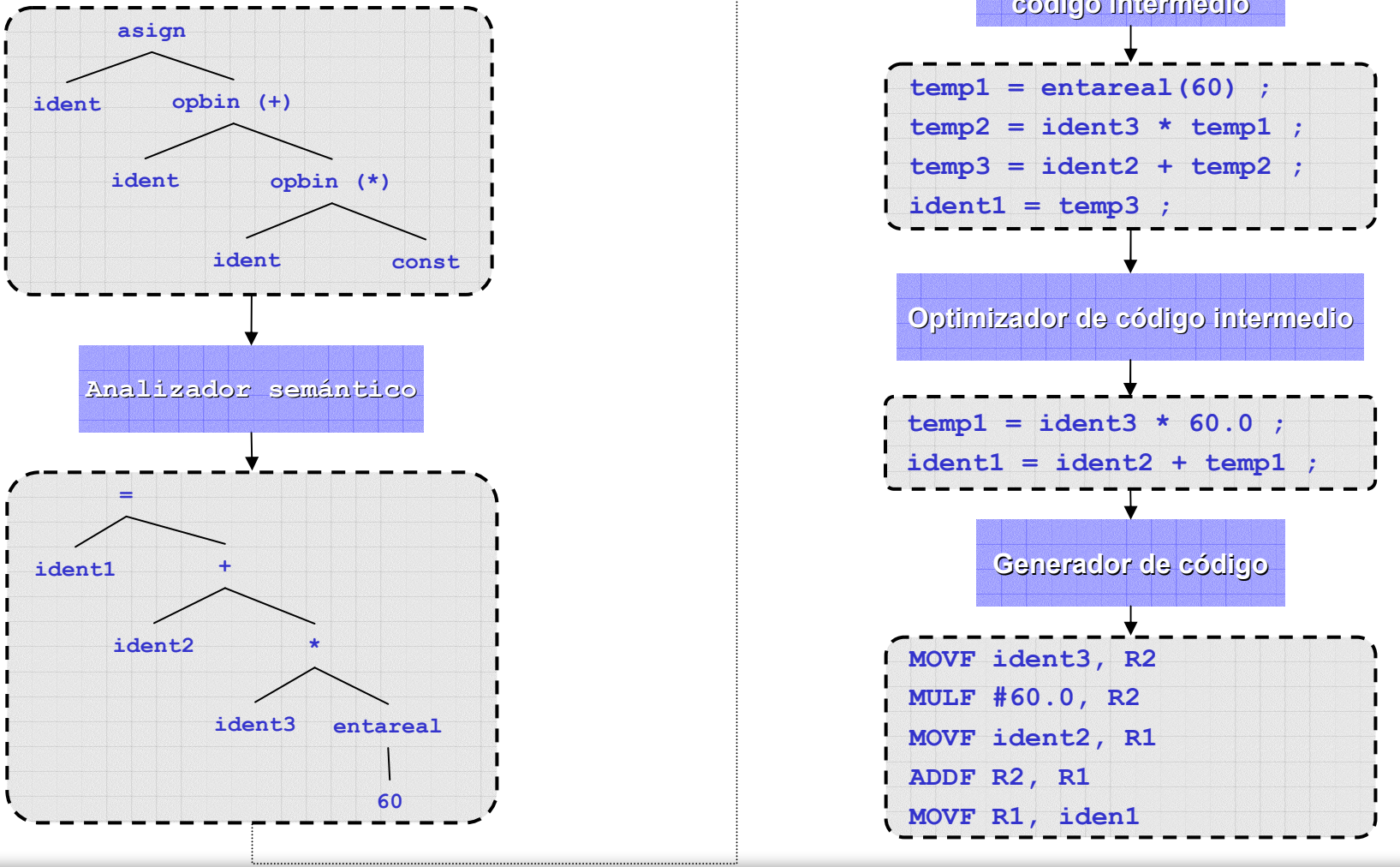
1.1.4 Modelo general de un traductor



1.1.4 Modelo general de un traductor: Ejemplo de traducción



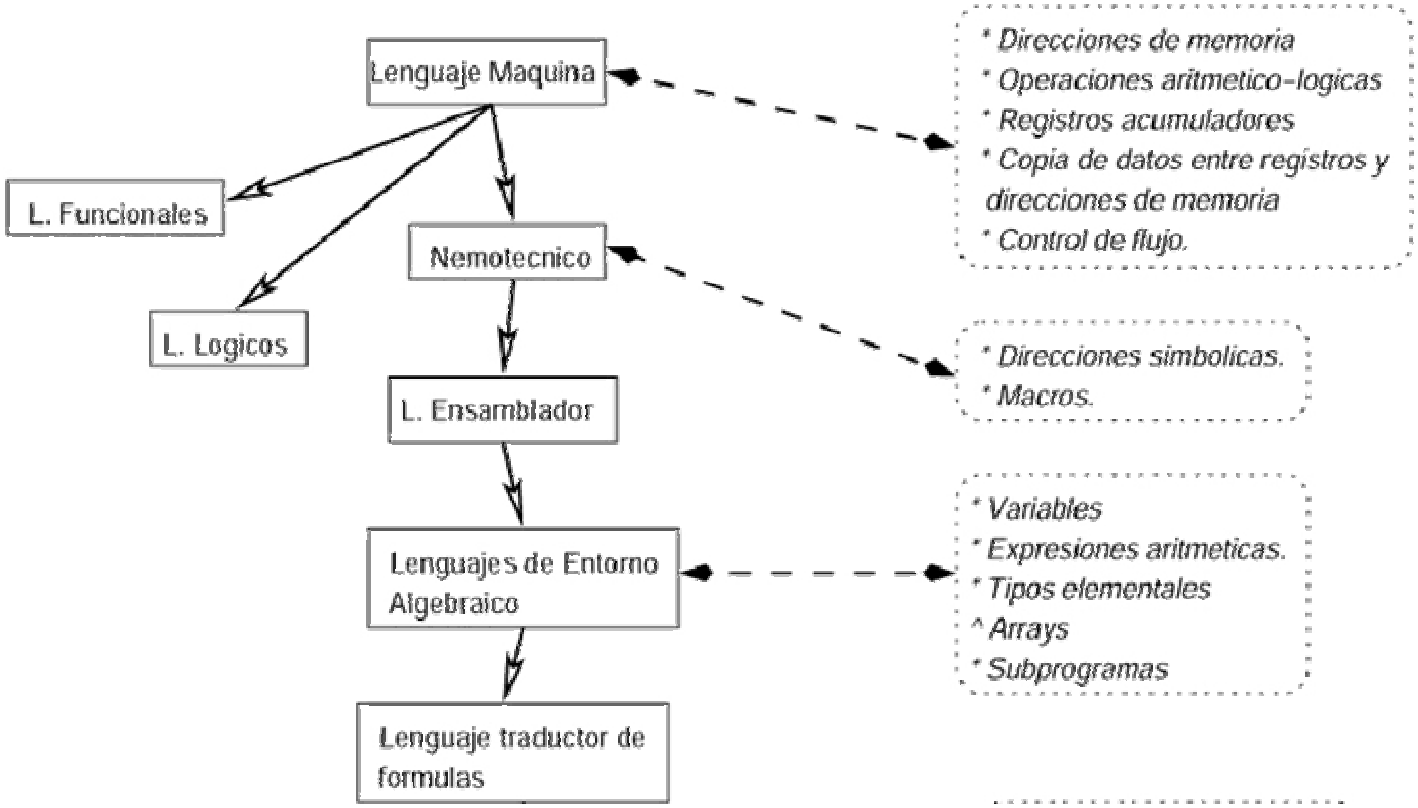
1.1.4 Modelo general de un traductor: Ejemplo de traducción



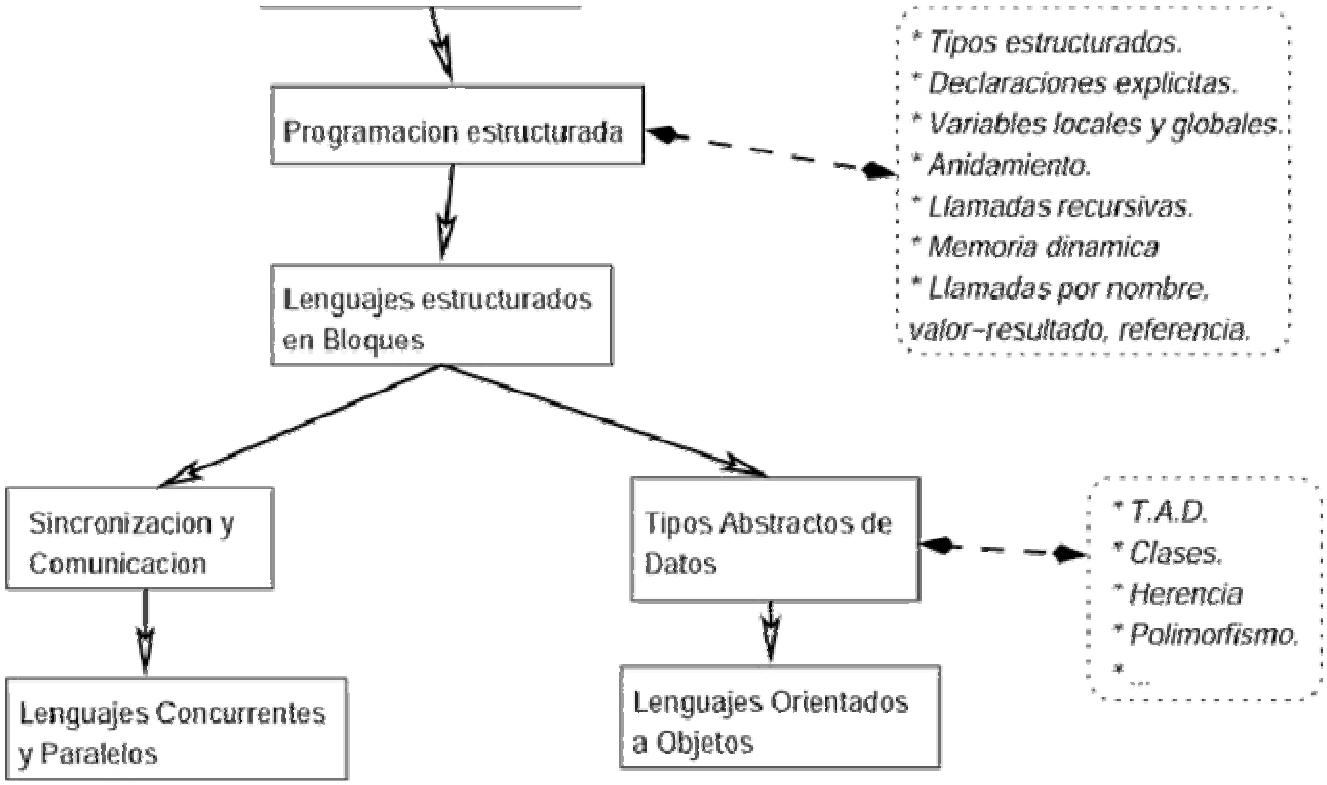
Tema 1 | Procesadores de Lenguajes

Contenidos

1.4.1 Evolución de los lenguajes de programación



1.4.1 Evolución de los lenguajes de programación



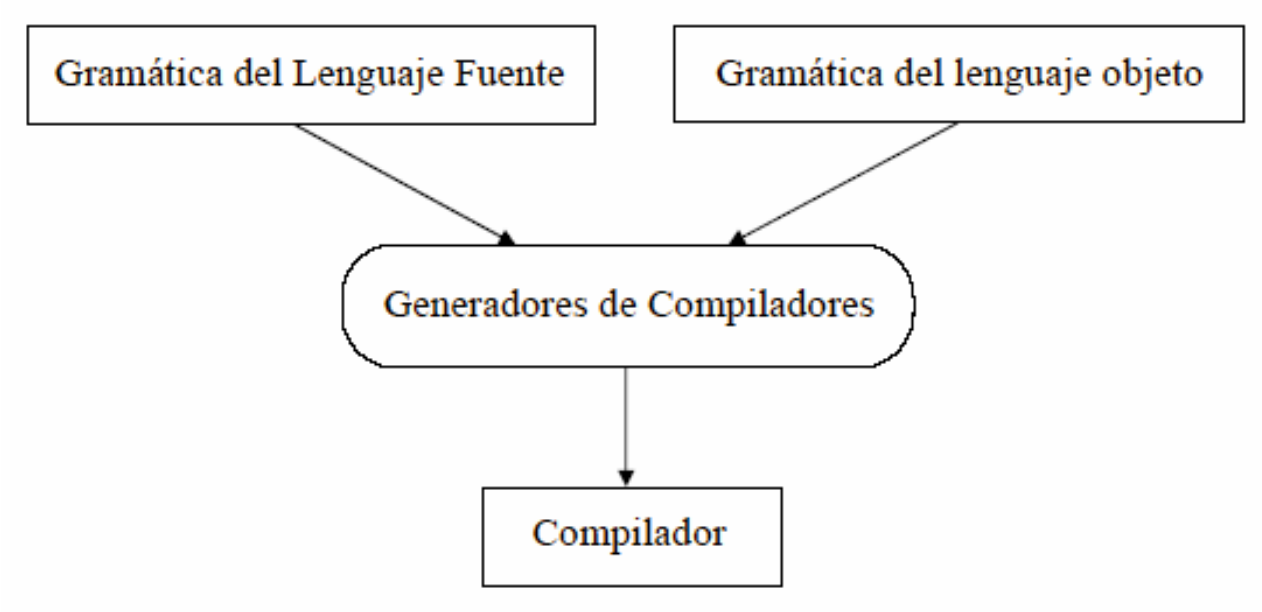
1.4.2 Evolución de los traductores

- **Década de los 50**
  1. Examen del texto de entrada
  2. Construcción de la tabla de direcciones y símbolos
  3. Traducción de direcciones simbólicas en direcciones de código
  4. Generación de código

- **Década de los 60**
  1. Análisis de léxico
     - Gramáticas regulares
     - Gramáticas libre de contexto
  2. Análisis semántico
  3. Generación de código

1.4.2 Evolución de los traductores

- **Generadores de compiladores.** Traductores dirigidos por sintaxis.



1.4.2 Evolución de traductores
Incidencia de conceptos de lenguajes de programación en traductores

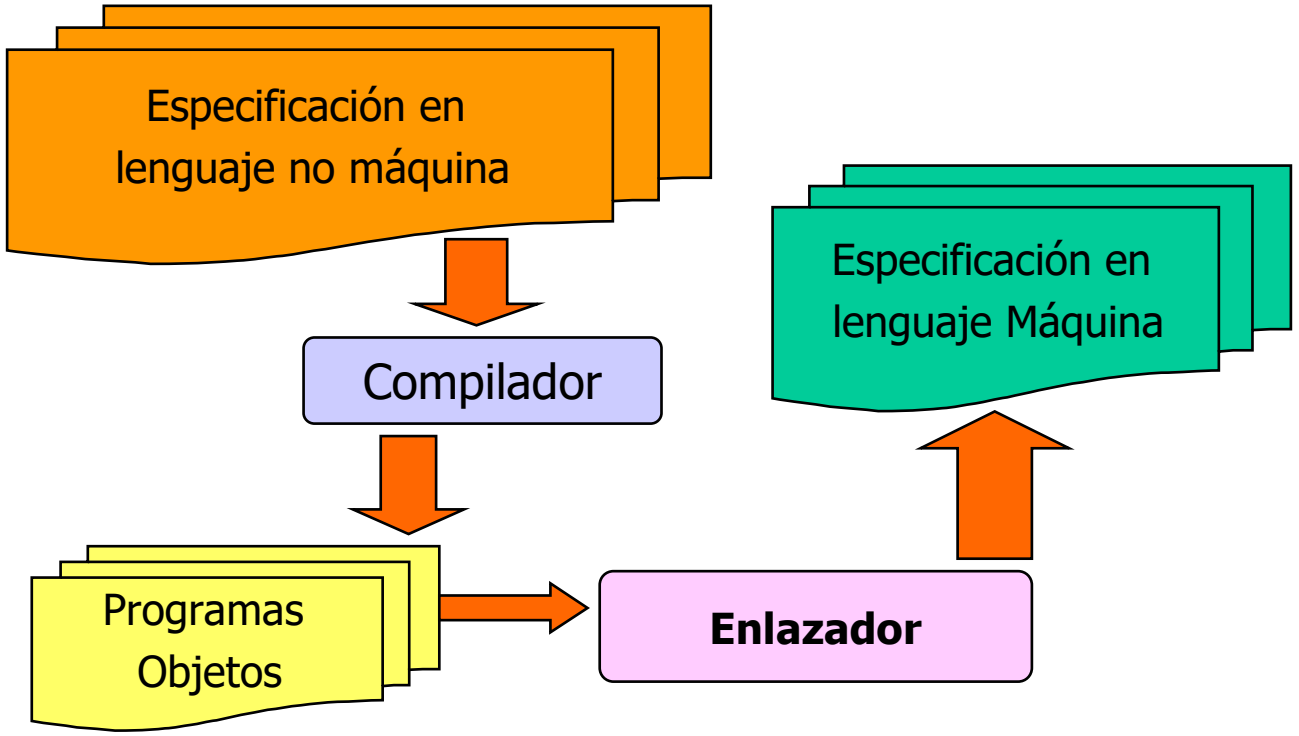
- **Lenguajes de entorno algebraico**
  - **Variables:** Usar mapa de memoria estática
  - **Transferencia de argumentos por referencia:** Usar tabla de símbolos

1.4.2 Evolución de los traductores
Incidencia de conceptos de lenguajes de programación en traductores

- **Lenguajes de programación estructurada**
  - Tipos estructurados
  - Declaraciones explícitas
  - Variables locales y globales: Usar tabla de símbolos por bloques
  - Anidamiento: Usar tabla de símbolos por bloques
  - Dimensiones dinámicas: Gestión dinámica de memoria
  - Subprogramas recursivos: Gestión dinámica de memoria
  - Transferencia de argumentos: Usar tabla de símbolos por bloques

1.4.3 Compiladores

- Un compilador obtiene una especificación en lenguaje máquina equivalente.



1.4.3 Compiladores

- Compilador

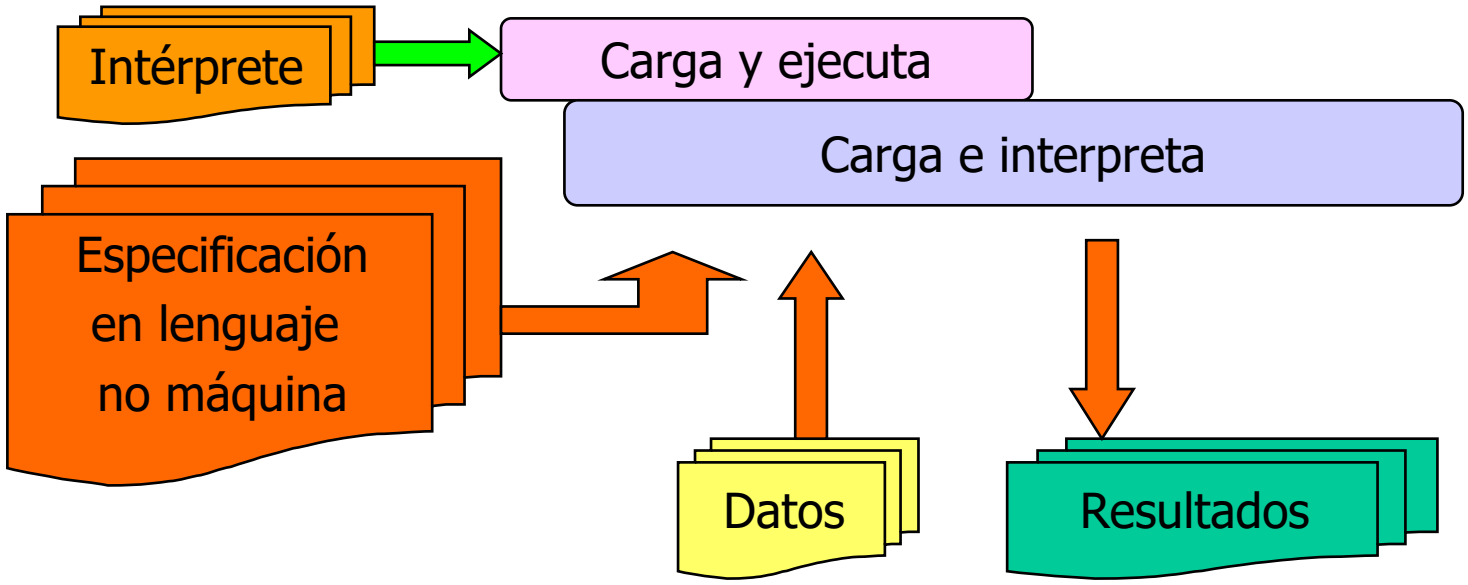
  Traduce la especificación de entrada a lenguaje máquina incompleto y con instrucciones máquina incompletas.

- Enlazador

  Enlaza los programas objetos y completa las instrucciones máquina incompletas, generando un ejecutable.

1.4.4 Intérpretes

- Un intérprete carga la especificación de un programa en lenguaje no máquina y la interpreta y ejecuta, instrucción a instrucción.



Tema 1 | Procesadores de Lenguajes

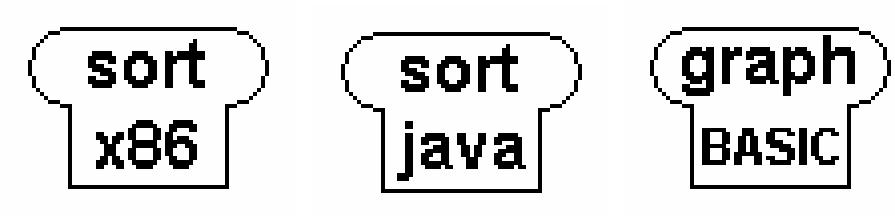
Contenidos

1.5 Arquitectura de procesadores de lenguajes

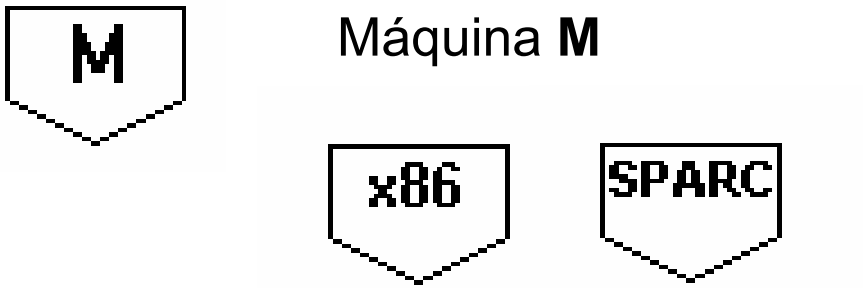
- Los traductores son programas que manipulan otros programas
- Hay tres lenguajes involucrados
  - Lenguaje fuente: **S** (*source*)
  - Lenguaje objeto: **T** (*target*)
  - Lenguaje de implementación del traductor: **L** (*language*)

1.5 Arquitectura de procesadores de lenguajes

- **Representación "TOMBSTONE" ("Lápida sepulcral", D. A. Watt, 1993)**
  - Diagrama para un programa

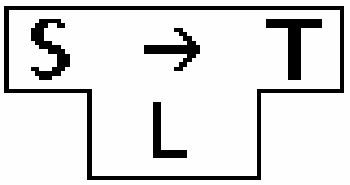


  - Diagrama para una máquina

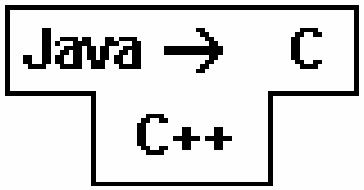


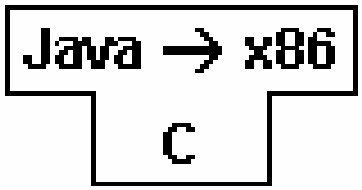
1.5 Arquitectura de procesadores de lenguajes

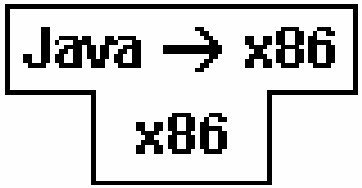
- **Representación "TOMBSTONE" ("Lápida sepulcral", D. A. Watt, 1993)**
  - Diagrama para un traductor



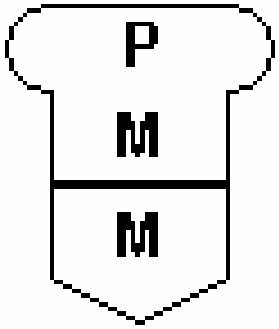
Traductor de lenguaje **S** a lenguaje **T** implementado en lenguaje **L**



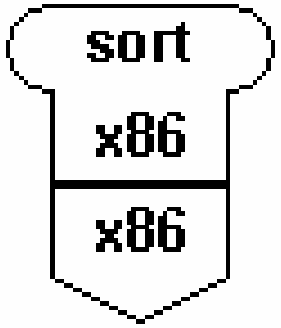


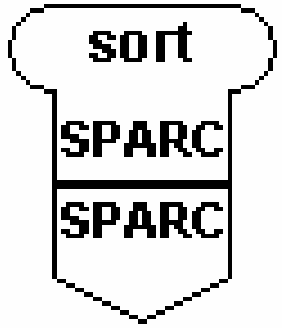


  - Diagrama para un programa en ejecución



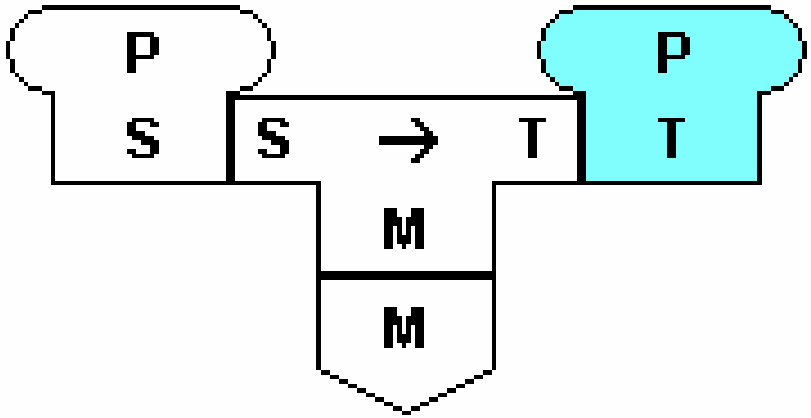
Programa **P** implementado en lenguaje **M** ejecutándose en máquina **M**



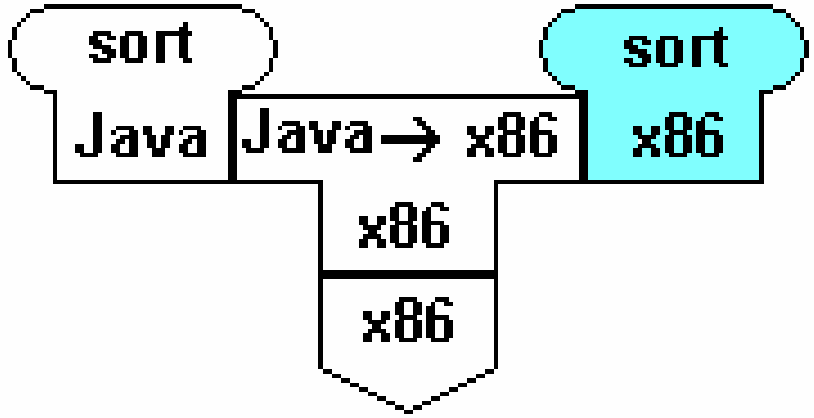


1.5 Arquitectura de procesadores de lenguajes

- **Representación "TOMBSTONE" ("Lápida sepulcral", D. A. Watt, 1993)**
  - Diagrama para un traductor en ejecución



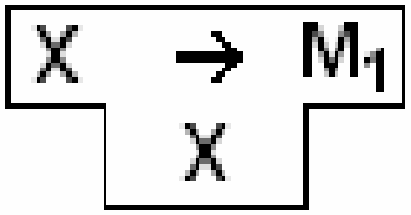
Traducción de un programa **P** implementado en lenguaje **S** a un programa **P** implementado en lenguaje **T**, usando un traductor de **S** a **T** implementado en lenguaje **M**, ejecutándose en máquina **M**



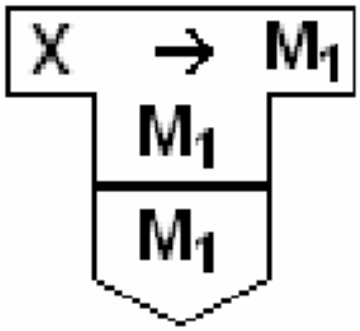
Traducción del programa **sort** implementado en **Java** a programa **sort** implementado en **x86**, usando traductor de **Java** a **x86** implementado en lenguaje **x86** y ejecutándose en máquina **x86**

1.5 Arquitectura de procesadores de lenguajes

- **¿Cómo portar un traductor de una máquina M_1 a una máquina M_2?**
  - Compilador existente que queremos portar

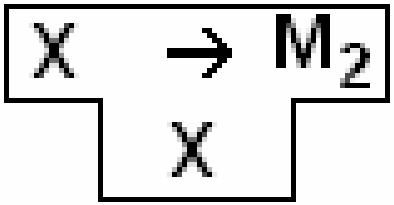


  - Disponemos de un compilador de lenguaje **X** que se ejecuta en máquina M_1 y produce código para máquina M_1

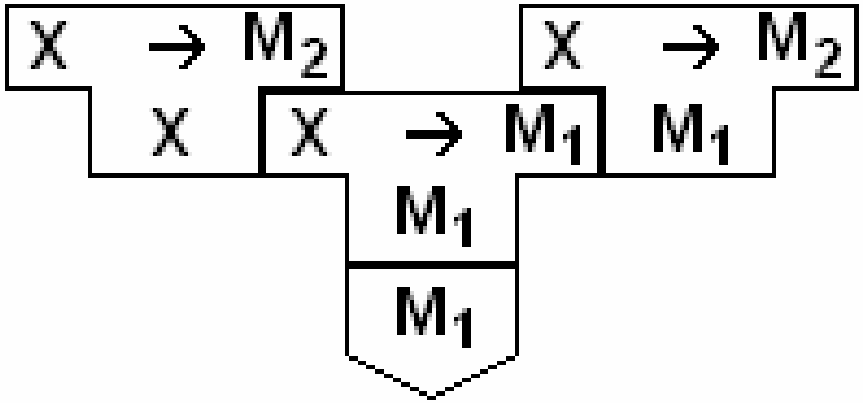


1.5 Arquitectura de procesadores de lenguajes

- **¿Cómo portar un traductor de una máquina M_1 a una máquina M_2?**
- **Solución: usar técnica *Half Bootstrap***
  - **Paso 1**: Reescribir la parte de generación de código del compilador existente para que produzca código para la nueva máquina **M_2**

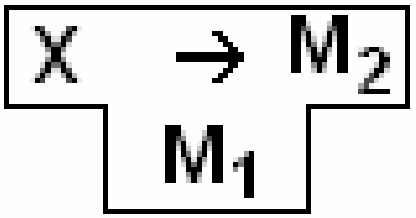


  - **Paso 2**: Compilar en la primera máquina el compilador modificado

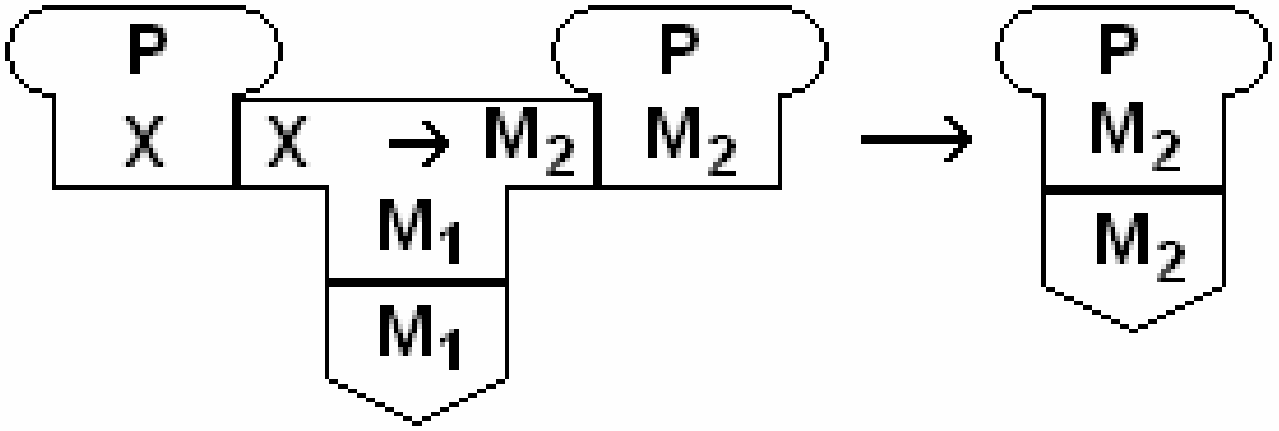


1.5 Arquitectura de procesadores de lenguajes

- **¿Cómo portar un traductor de una máquina M_1 a una máquina M_2?**
- **Solución: usar técnica *Half Bootstrap***
  - **Resultado**: Compilador que se ejecuta en máquina **M_1** y produce código para máquina **M_2 (compilador cruzado)**:

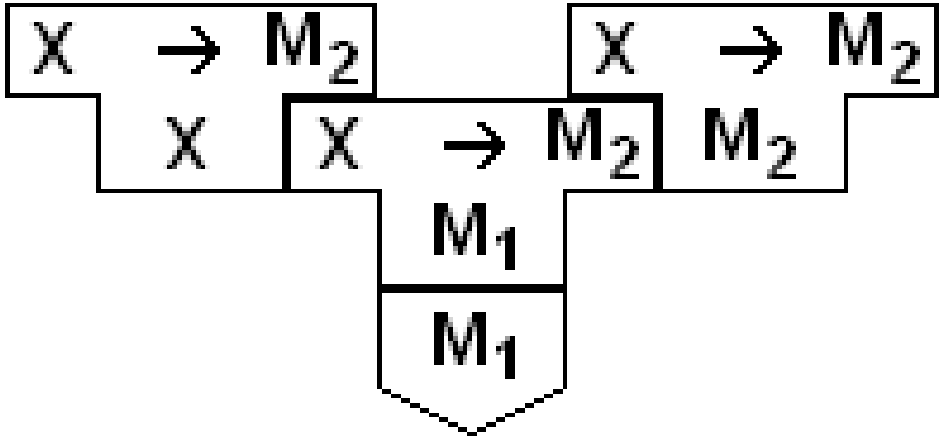


  - Podemos ejecutar en máquina **M_2** programas escritos en lenguaje **X**, compilados en máquina **M_1**:



1.5 Arquitectura de procesadores de lenguajes

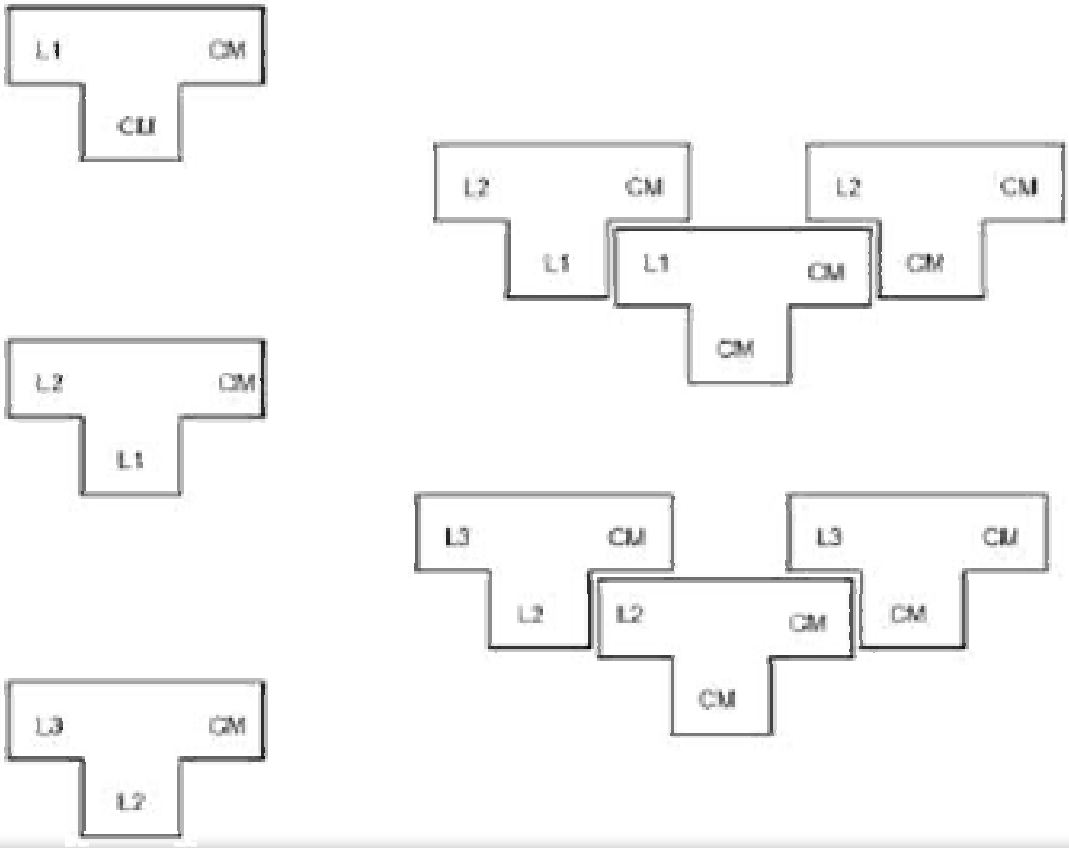
- **¿Cómo portar un traductor de una máquina M_1 a una máquina M_2?**
- **Solución: usar técnica *Half Bootstrap***
  - **Resultado**: El compilador cruzado puede ser usado para compilarse a sí mismo, produciendo un compilador que se puede transferir a máquina **M_2**:



1.5 Arquitectura de procesadores de lenguajes

- **Bootstrapping**: Traducir un lenguaje complejo usando un lenguaje simple

$$L1 \subset L2 \subset L3$$



1.5 Arquitectura de procesadores de lenguajes

- **Ejemplo de Bootstrapping**
  - Compilación de un compilador de C escrito en ensamblador