UNIVERSIDAD DE CÓRDOBA

ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE INFORMÁTICA Y ANÁLISIS NUMÉRICO

PROCESADORES DE LENGUAJES

GRADO EN INGENIERÍA INFORMÁTICA

ESPECIALIDAD DE COMPUTACIÓN

TERCER CURSO

SEGUNDO CUATRIMESTRE

PROGRAMA

TEMA I.- INTRODUCCIÓN

TEMA II.- ANÁLISIS LEXICOGRÁFICO

TEMA III.- FUNDAMENTOS TEÓRICOS DEL ANÁLISIS SINTÁCTICO

TEMA IV.- ANÁLISIS SINTÁCTICO DESCENDENTE

TEMA V.- ANÁLISIS SINTÁCTICO ASCENDENTE

TEMA I.- INTRODUCCIÓN

- TRADUCCIÓN E INTEPRETACIÓN
- TIPOS DE TRADUCTORES
- PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN
- ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS
- HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES
- COMBINACIÓN DE COMPILADORES: “BOOTSTRAPPING”

TEMA I.- INTRODUCCIÓN

- TRADUCCIÓN E INTEPRETACIÓN
- TIPOS DE TRADUCTORES
- PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN
- ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS
- HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES
- COMBINACIÓN DE COMPILADORES: “BOOTSTRAPPING”

- **TRADUCCIÓN E INTEPRETACIÓN**

✓Los **algoritmos** permiten **resolver** los **problemas** de computación

- **TRADUCCIÓN E INTEPRETACIÓN**

✓Los **algoritmos** permiten **resolver** los **problemas** de computación

✓ **Programa fuente**: algoritmo escrito en un **lenguaje de programación**

- **TRADUCCIÓN E INTEPRETACIÓN**

✓Los **algoritmos** permiten **resolver** los **problemas** de computación

✓ **Programa fuente**: algoritmo escrito en un **lenguaje de programación**

✓ Los **programas fuentes no** pueden ser **ejecutados** directamente por los **ordenadores**

- **TRADUCCIÓN E INTEPRETACIÓN**

✓Los **algoritmos** permiten **resolver** los **problemas** de computación

✓ **Programa fuente**: algoritmo escrito en un **lenguaje de programación**

✓ Los **programas fuentes no** pueden ser **ejecutados** directamente por los **ordenadores**

✓ Los **ordenadores** sólo **ejecutan código** escrito en **lenguaje máquina**

- **TRADUCCIÓN E INTEPRETACIÓN**

✓Los **algoritmos** permiten **resolver** los **problemas** de computación

✓ **Programa fuente**: algoritmo escrito en un **lenguaje de programación**

✓ Los **programas fuentes no** pueden ser **ejecutados** directamente por los **ordenadores**

✓ Los **ordenadores** sólo **ejecutan código** escrito en **lenguaje máquina**

✓ **Problema**: transformar el programa fuente en código ejecutable

- **TRADUCCIÓN E INTEPRETACIÓN**

✓Los **algoritmos** permiten **resolver** los **problemas** de computación

✓ **Programa fuente**: algoritmo escrito en un **lenguaje de programación**

✓ Los **programas fuentes no** pueden ser **ejecutados** directamente por los **ordenadores**

✓ Los **ordenadores** sólo **ejecutan código** escrito en **lenguaje máquina**

✓ **Problema**: transformar el programa fuente en código ejecutable

Programa fuente → **Transformador** → **Código ejecutable**

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ Existen **dos tipos** de **transformación**:

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ Existen **dos tipos** de **transformación**:
    - ➢ **Traducción**
    - ➢ **Interpretación**

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ Existen **dos tipos** de **transformación**:
    - ➢ **Traducción**
      - ❑ Un **programa fuente** (**alto nivel**) es **convertido** en **código ejecutable** (**bajo nivel**) que puede ser **ejecutado independientemente.**

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Traducción**

Programa fuente → **Traductor**

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Traducción**

Programa fuente → **Traductor**

↓

Errores de traducción

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Traducción**

Programa fuente → **Traductor** → **Código ejecutable**

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Traducción**

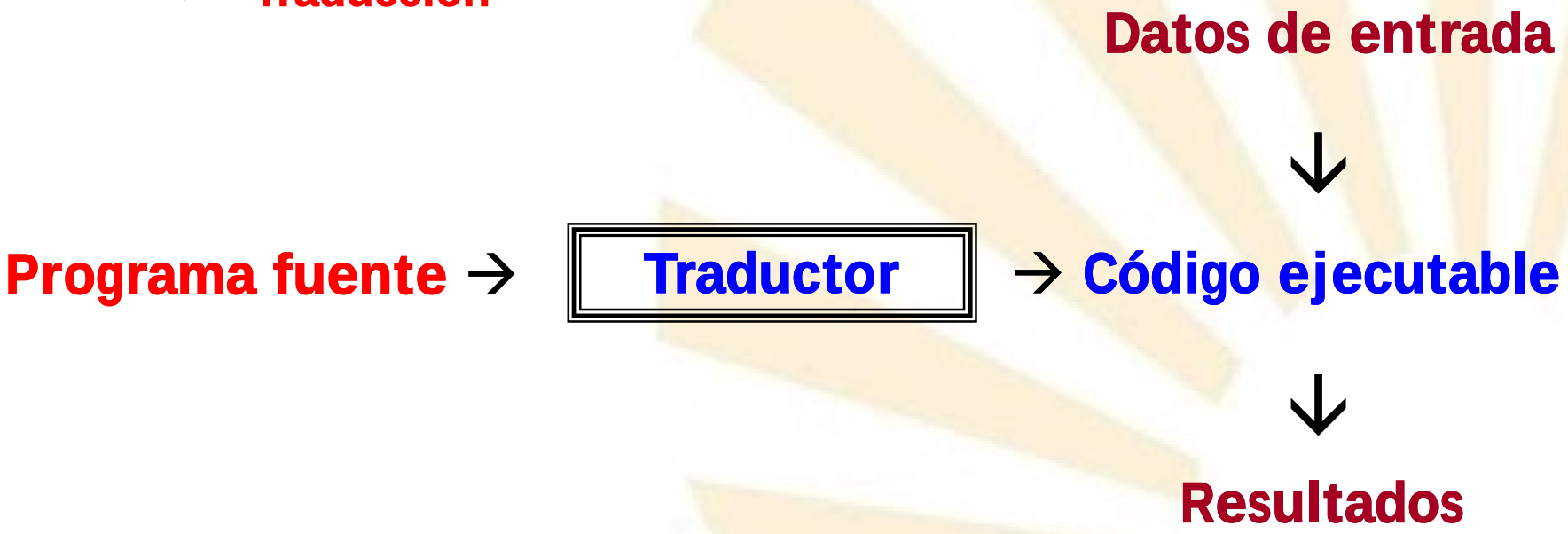
Datos de entrada

↓

Programa fuente → **Traductor** → **Código ejecutable**

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Traducción**

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Traducción**



- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ Existen **dos tipos** de **transformaciones**
    - ➢ **Traducción**
    - ➢ **Interpretación** o **simulación**: consta de **tres fases** que se repiten sucesivamente

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ Existen **dos tipos** de **transformaciones**
    - ➢ **Traducción**
    - ➢ **Interpretación** o **simulación**: consta de **tres fases** que se repiten sucesivamente
      1. **Análisis** del código fuente para determinar la siguiente sentencia a ejecutar.

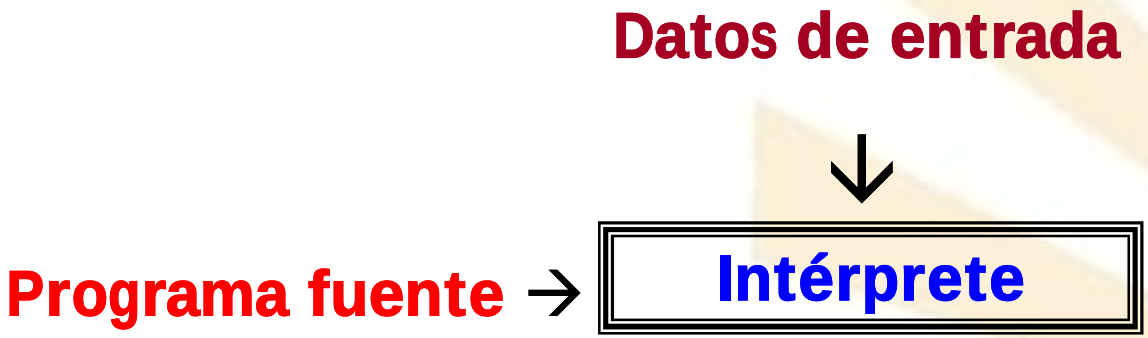
- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ Existen **dos tipos** de **transformaciones**
    - ➢ **Traducción**
    - ➢ **Interpretación** o **simulación**: consta de **tres fases** que se repiten sucesivamente
      1. **Análisis** del código fuente para determinar la siguiente sentencia a ejecutar.
      2. **Generación** del código que se ha de ejecutar.

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ Existen **dos tipos** de **transformaciones**
    - ➢ **Traducción**
    - ➢ **Interpretación** o **simulación**: consta de **tres fases** que se repiten sucesivamente
      1. **Análisis** del código fuente para determinar la siguiente sentencia a ejecutar.
      2. **Generación** del código que se ha de ejecutar.
      3. **Ejecución** del código generado.

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Interpretación**

Programa fuente → **Intérprete**

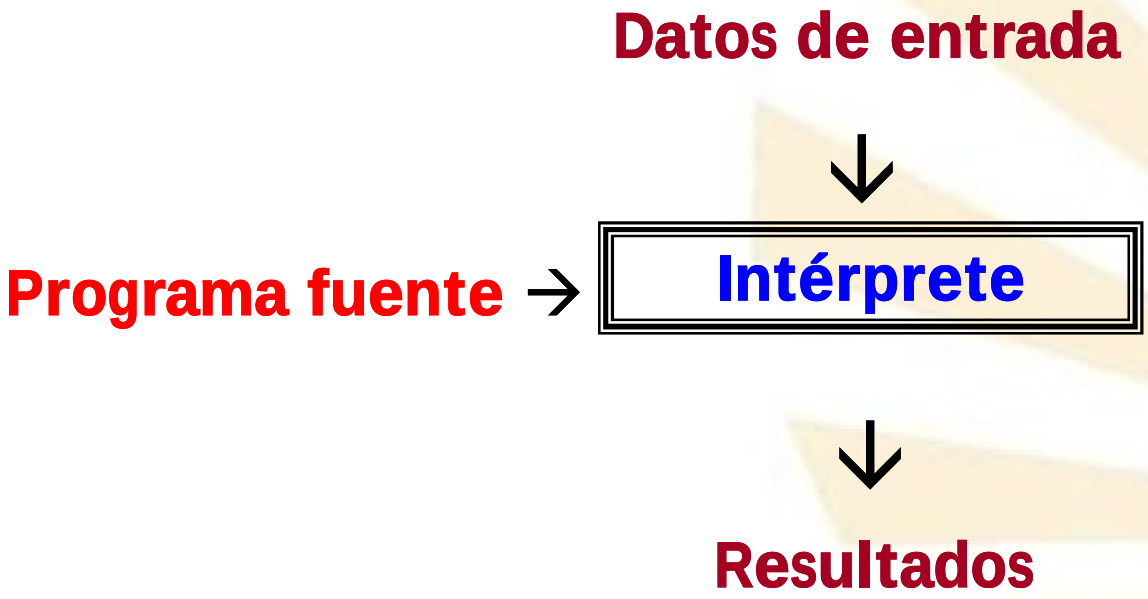
- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Interpretación**



- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Interpretación**

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Interpretación**

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Interpretación**



- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Diferencias** fundamentales entre **traducción** e **interpretación**

- **TRADUCCIÓN E INTEPRETACIÓN**

  ✓ **Diferencias** fundamentales entre **traducción** e **interpretación**

➢ **Traducción**

❑ **Independencia**

- El **código generado** se puede **ejecutar independientemente** del programa fuente y del traductor.
- Se **traduce una vez** y se **ejecuta muchas veces**.

➢ **Interpretación**

❑ **Dependencia**

- El código generado sólo se puede **ejecutar** con el **intérprete** y el **programa fuente**.
- Se interpreta y ejecuta a **la vez**.

- **TRADUCCIÓN E INTEPRETACIÓN**

✓ **Diferencias** fundamentales entre **traducción** e **interpretación**

➢ **Traducción**

❑ **Independencia**

❑ **Necesidades de memoria**

▪ El **código** generado se ha de **almacenar** en **memoria**.

➢ **Interpretación**

❑ **Dependencia**

❑ **Sin necesidad de memoria**

▪ El **código** generado **no se almacena** en **memoria**.

- **TRADUCCIÓN E INTEPRETACIÓN**

  ✓ **Diferencias** fundamentales entre **traducción** e **interpretación**

➢ **Traducción**

- ❑ **Independencia**
- ❑ **Necesidades de memoria**
- ❑ **Eficiencia**
  - Una vez generado el **código**, éste **se ejecuta con rapidez**.

➢ **Interpretación**

- ❑ **Dependencia**
- ❑ **Sin necesidad de memoria**
- ❑ **Menos eficiencia**
  - El **código** se ha de **volver a generar** para volver a ser ejecutado.

- **TRADUCCIÓN E INTEPRETACIÓN**

✓ **Diferencias** fundamentales entre **traducción** e **interpretación**

➢ **Traducción**

- ❑ **Independencia**
- ❑ **Necesidades de memoria**
- ❑ **Eficiencia**
- ❑ **Global**
  - Posee una **visión completa** del **programa** pudiendo generar **mensajes de error** más **detallados.**

➢ **Interpretación**

- ❑ **Dependencia**
- ❑ **Sin necesidad de memoria**
- ❑ **Menos eficiencia**
- ❑ **Local**
  - Posee una **visión parcial** del programa, ya que interpreta el código sentencia a sentencia.

- **TRADUCCIÓN E INTEPRETACIÓN**

✓ **Diferencias** fundamentales entre **traducción** e **interpretación**

➢ **Traducción**

- ❑ **Independencia**
- ❑ **Necesidades de memoria**
- ❑ **Eficiencia**
- ❑ **Global**
- ❑ **No interactividad**
  - **No** permite la **interacción** con el programa fuente

➢ **Interpretación**

- ❑ **Dependencia**
- ❑ **Sin necesidad de memoria**
- ❑ **Menos eficiencia**
- ❑ **Local**
- ❑ **Interactividad**
  - Permite una **interacción** con el **programa durante** su **desarrollo**.

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Diferencias** fundamentales entre **traducción** e **interpretación**

➢ **Traducción**
- ❑ **Independencia**
- ❑ **Necesidades de memoria**
- ❑ **Eficiencia**
- ❑ **Global**
- ❑ **No interactividad**
- ❑ **No inclusión de código durante la ejecución**

➢ **Interpretación**
- ❑ **Dependencia**
- ❑ **Sin necesidad de memoria**
- ❑ **Menos eficiencia**
- ❑ **Local**
- ❑ **Interactividad**
- ❑ **Inclusión de código durante la ejecución**
  - v.g.: intérpretes de Smalltalk, Lisp y Prolog.

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Combinación** de la **traducción** e **interpretación**:
    - ➢ son procesos complementarios

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Combinación** de la **traducción** e **interpretación**:
    - ➢ **Interpretación + traducción**
      - ❑ Se facilita la **depuración del código**:
        - ❖ la interpretación permite la interacción con el programa durante su desarrollo.
      - ❑ El código depurado permite **generar código** ejecutable **más eficiente**.

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Combinación** de la **traducción** e **interpretación**:
    - ➢ **Interpretación + traducción**:
      - ❑ Se facilita la **depuración**:
        - ❖ la interpretación permite la interacción con el programa durante su desarrollo.
      - ❑ El código depurado permite **generar código** ejecutable más **eficiente**.
    - ➢ **Traducción + interpretación**:
      - ❑ El programa fuente se traduce a **código intermedio**.
      - ❑ El **código intermedio** puede ser **interpretado** en diferentes entornos de ejecución.
      - ❑ V.g.: Java, C#, ...

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Tipos de lenguajes**

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Tipos de lenguajes**
    - ➢ **Lenguajes interpretados**:
      - ❑ Utilizan un **intérprete** para ejecutar sus programas.
      - ❑ V.g.: APL, Lisp, Scheme, Prolog, Java, Smalltalk, etc.

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Tipos de lenguajes**
    - ➢ **Lenguajes interpretados**:
      - ❑ Utilizan un **intérprete** para ejecutar sus programas.
    - ➢ **Lenguajes compilados**:
      - ❑ Utilizan un traductor denominado "**compilador**" para generar el programa ejecutable.
      - ❑ V.g.: Fortran, Pascal, Ada, C, C++

- **TRADUCCIÓN E INTEPRETACIÓN**
  - ✓ **Tipos de lenguajes**
    - ➢ **Lenguajes interpretados**:
      - ❑ Utilizan un **intérprete** para ejecutar sus programas.
    - ➢ **Lenguajes compilados**:
      - ❑ Utilizan un traductor denominado "**compilador**" para generar el programa ejecutable.
    - ➢ **No** es una **clasificación excluyente**: existen lenguajes que poseen intérpretes y compiladores:
      - ❑ Intérprete: utilizado para el desarrollo, depuración y puesta a punto.
      - ❑ Compilador: genera el programa ejecutable.
      - ❑ V. g.: Visual Basic, Builder C++, etc.

TEMA I.- INTRODUCCIÓN

- TRADUCCIÓN E INTEPRETACIÓN
- TIPOS DE TRADUCTORES
- PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN
- ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS
- HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES
- COMBINACIÓN DE COMPILADORES: “BOOTSTRAPPING”

- **TIPOS DE TRADUCTORES**
  - ✓ **Preprocesador**
  - ✓ **Compilador**
  - ✓ **Ensamblador**
  - ✓ **Enlazador (“linker”)**
  - ✓ **Cargador (“loader”)**

- **TIPOS DE TRADUCTORES**

✓ **Preprocesador**

➢ Programa **inicial** escrito en un lenguaje de **alto nivel extendido**

➢ Programa **final** escrito en un lenguaje de **alto nivel estándar**

➢ V.g.: **"cpp"** es un preprocesador del lenguaje C que realiza las siguientes acciones:

- ❑ **Expandir macros:** #define PI 3.141592
- ❑ **Incluir ficheros:** #include <stdio.h>
- ❑ **Eliminar comentarios:** /* Menú principal */
- ❑ **Etc.**

➢ **Nota**: también existen preprocesadores para **embellecer** el programa fuente.

- **TIPOS DE TRADUCTORES**
  - ✓ **Preprocesador**
  - ✓ **Compilador**
    - ➢ Programa **inicial** escrito en un lenguaje de **alto nivel**
    - ➢ Programa **final** escrito en un lenguaje de **bajo nivel (máquina o ensamblador).**

- **TIPOS DE TRADUCTORES**
  - ✓ **Preprocesador**
  - ✓ **Compilador**
  - ✓ **Ensamblador**
    - ➢ Programa **inicial** escrito en lenguaje **ensamblador**
    - ➢ Programa **final** escrito en **código máquina**
    - ➢ **Nota**: el ensamblador es un caso particular de compilador.

- **TIPOS DE TRADUCTORES**
  - ✓ **Preprocesador**
  - ✓ **Compilador**
  - ✓ **Ensamblador**
  - ✓ **Enlazador (“linker”)**
    - ➢ Programa **inicial** escrito en **código reubicable** (posiciones de memoria relativas)
    - ➢ Programa **final** escrito en **código máquina absoluto o ejecutable**
    - ➢ **Notas**:
      - ❑ Además incluye el código de las funciones de las **bibliotecas** utilizadas por el programa fuente.
      - ❑ Algunas veces genera código reubicable.

- **TIPOS DE TRADUCTORES**
  - ✓ **Preprocesador**
  - ✓ **Compilador**
  - ✓ **Ensamblador**
  - ✓ **Enlazador**
  - ✓ **Cargador (“loader”):**
    - ➢ Programa **inicial** escrito en **código reubicable**
    - ➢ Programa **final** escrito en **código máquina ejecutable**
    - ➢ **Nota**: **no** suele ser un programa **independiente**.

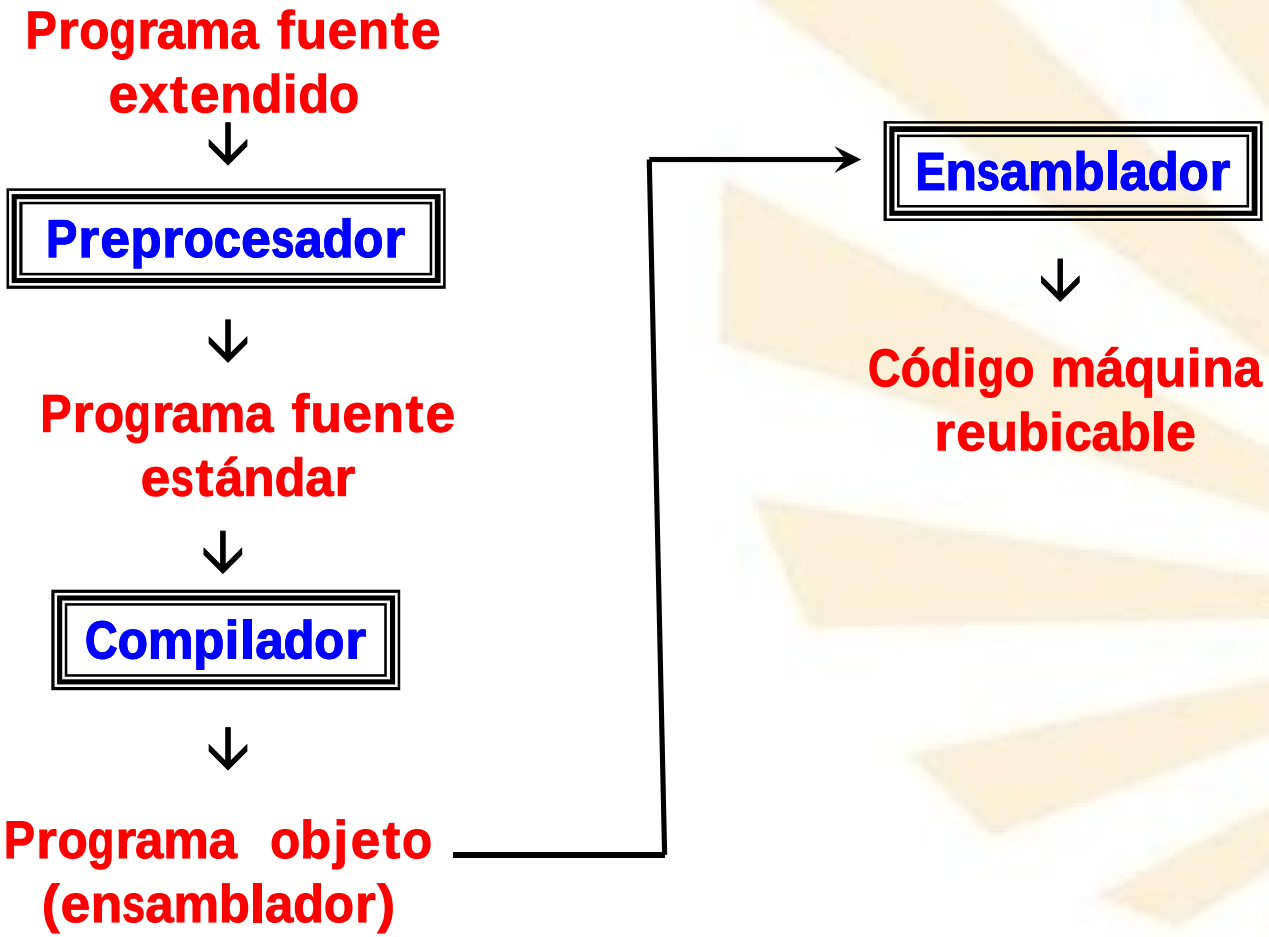
- **TIPOS DE TRADUCTORES**
  - ✓ **Combinación** de los tipos de traductores

Programa fuente extendido

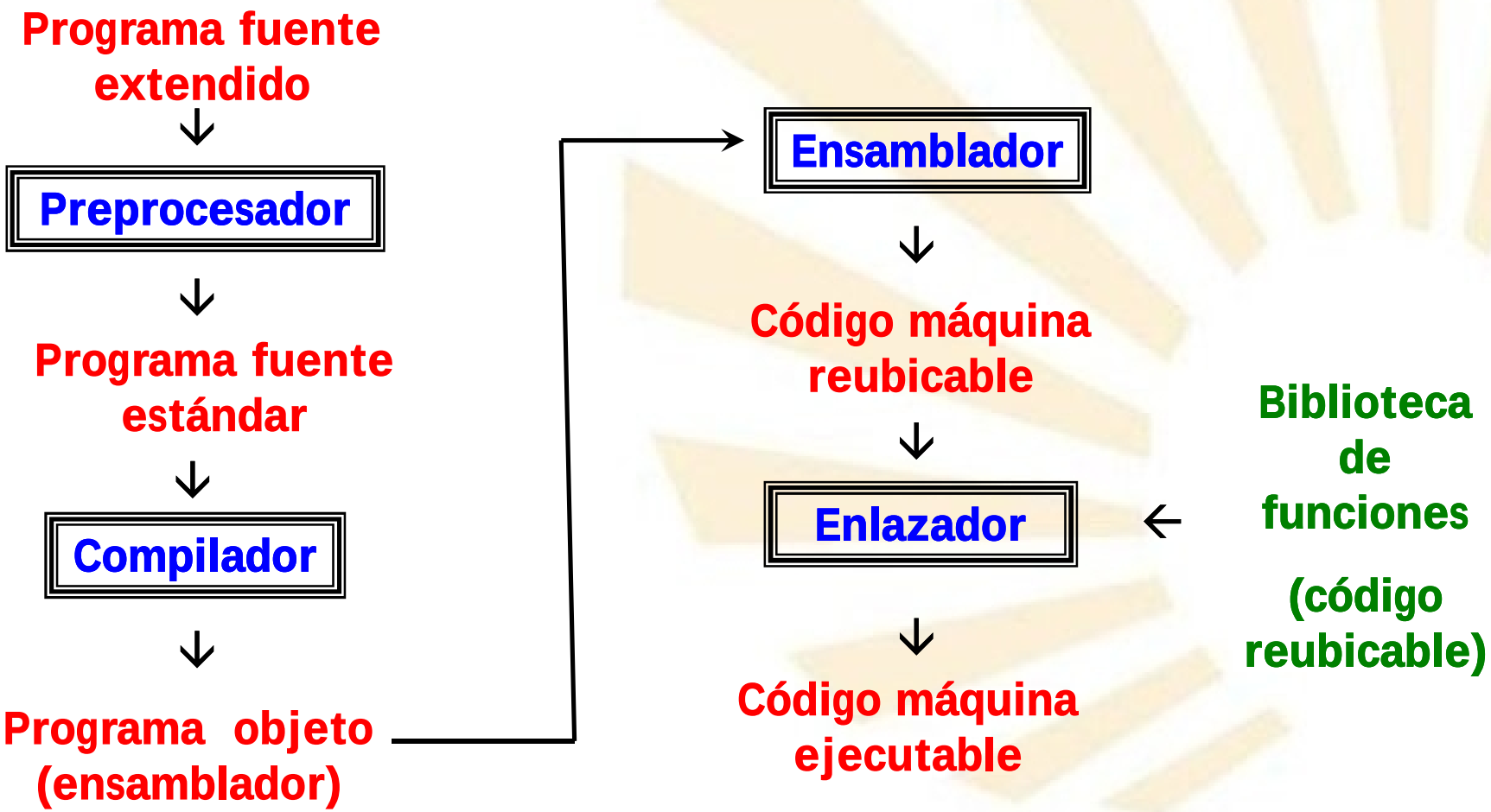
- **TIPOS DE TRADUCTORES**
  - ✓ **Combinación** de los tipos de traductores

- **TIPOS DE TRADUCTORES**
  - ✓ **Combinación** de los tipos de traductores

- **TIPOS DE TRADUCTORES**
  - ✓ **Combinación** de los tipos de traductores



- **TIPOS DE TRADUCTORES**
  - ✓ **Combinación** de los tipos de traductores



TEMA I.- INTRODUCCIÓN

- TRADUCCIÓN E INTEPRETACIÓN
- TIPOS DE TRADUCTORES
- PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN
- ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS
- HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES
- COMBINACIÓN DE COMPILADORES: “BOOTSTRAPPING”

- **PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN**
  - ✓ Editor basado en la estructura sintáctica del lenguaje de programación
  - ✓ Depurador
  - ✓ Generador del programa ejecutable
  - ✓ Perfilador
  - ✓ Entorno de desarrollo integrado

- **PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN**

  ✓ **Editor basado en la estructura sintáctica del lenguaje de programación**

  ➢ Facilita la **edición** de los programas al mostrar las **estructuras de las sentencias** de un lenguaje de programación.

  ➢ **Evita** la aparición de **errores léxicos** y, sobre todo, **sintácticos**.

- **PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN**

  ✓ **Editor basado en la estructura sintáctica del lenguaje de programación**

  ✓ **Depurador**

  ➢ En realidad es un **intérprete** que permite **ejecutar** el programa de forma **supervisada**.

  ➢ Permite la **ejecución paso a paso** del programa.

  ➢ Permite comprobar los valores de las variables, establecer puntos de parada, etc.

  ➢ V.g.: algunos depuradores de C son gdb, ddd, dbx, dbxtool.

- **PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN**

  ✓ **Editor basado en la estructura sintáctica del lenguaje de programación**

  ✓ **Depurador**

  ✓ **Generador del programa ejecutable**

  ➢ Analiza las **dependencias** del código las bibliotecas de **funciones** para crear el código ejecutable.

  ➢ V.g.: Install Shield, Setup Factory, etc.

- **PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN**

  ✓ **Editor basado en la estructura sintáctica del lenguaje de programación**

  ✓ **Depurador**

  ✓ **Generador del programa ejecutable**

  ✓ **Perfilador**

  ➢Herramienta muy útil para la **optimización** de los programas.

  ➢Permite conocer el perfil de ejecución de un programa.

  ➢Genera estadísticas sobre la ejecución del programa relativas a uso de funciones, accesos a memoria, tiempos de ejecución, etc.

  ➢ Se pueden descubrir "los cuellos de botella", es decir, dónde se requiere más tiempo de ejecución.

- **PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN**
  - ✓ **Editor basado en la estructura sintáctica del lenguaje de programación**
  - ✓ **Depurador**
  - ✓ **Generador del programa ejecutable**
  - ✓ **Perfilador**
  - ✓ **Entorno de desarrollo integrado**: incluye
    - un editor,
    - un compilador,
    - un enlazador,
    - un depurador,
    - etc.

TEMA I.- INTRODUCCIÓN

- TRADUCCIÓN E INTEPRETACIÓN
- TIPOS DE TRADUCTORES
- PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN
- ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS
- HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES
- COMBINACIÓN DE COMPILADORES: “BOOTSTRAPPING”

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
  - ✓ **Pasos**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Análisis:** se encarga de comprobar que el programa fuente está bien escrito
    - ➢ **Síntesis:** se ocupa de la generación del código ejecutable
    - ➢ **Componentes auxiliares**:
      - ❑ Administrador de la tabla de símbolos
      - ❑ Gestor de errores

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Análisis**
      - ❑ **Análisis léxico**
    - ➢ **Síntesis**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Análisis**
      - ❑ **Análisis léxico**
      - ❑ **Análisis sintáctico**
    - ➢ **Síntesis**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Análisis**
      - ❑ **Análisis léxico**
      - ❑ **Análisis sintáctico**
      - ❑ **Análisis semántico**
    - ➢ **Síntesis**

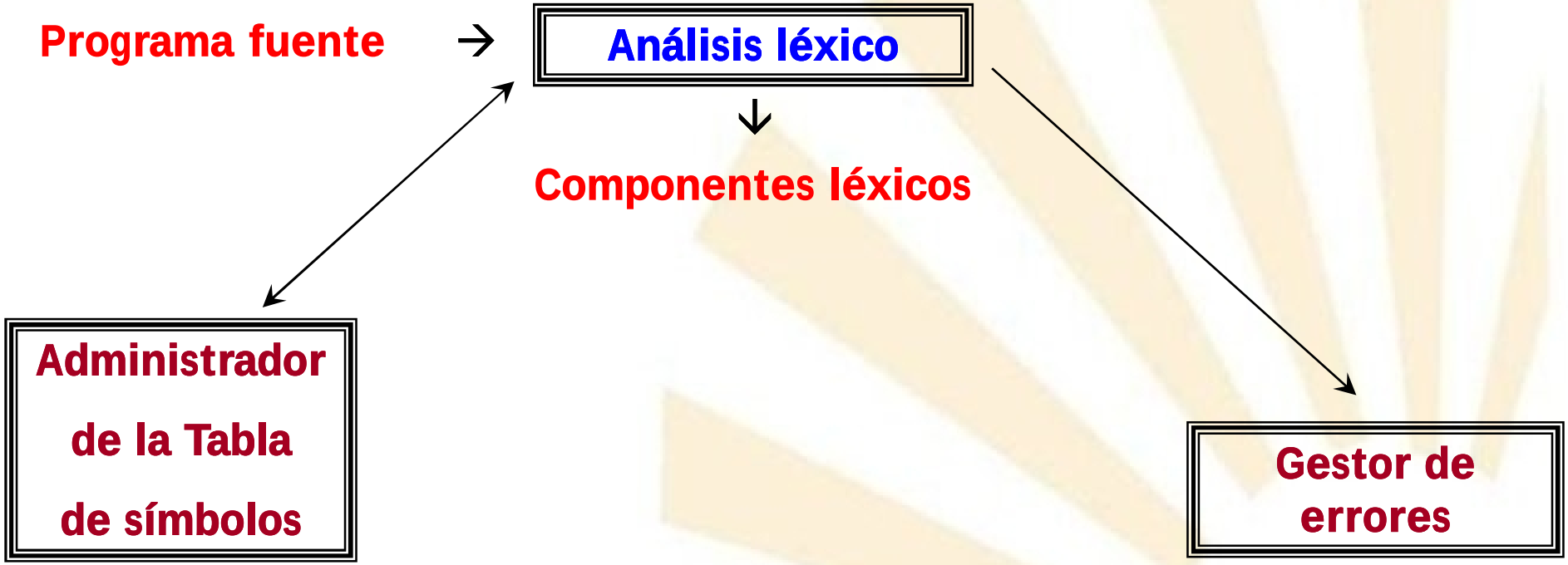
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Análisis**
      - ❑ **Análisis léxico**
      - ❑ **Análisis sintáctico**
      - ❑ **Análisis semántico**
    - ➢ **Síntesis**
      - ❑ **Generación de código intermedio**

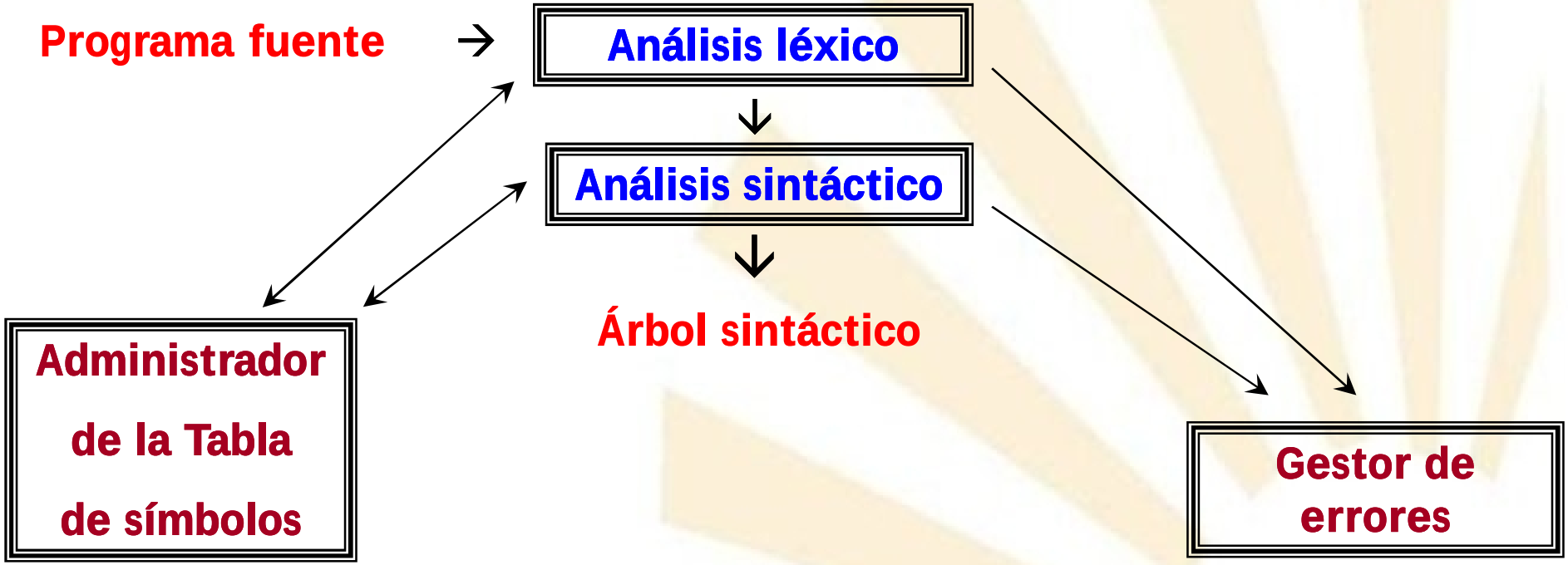
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Análisis**
      - ❑ **Análisis léxico**
      - ❑ **Análisis sintáctico**
      - ❑ **Análisis semántico**
    - ➢ **Síntesis**
      - ❑ **Generación de código intermedio**
      - ❑ **Optimización de código intermedio**

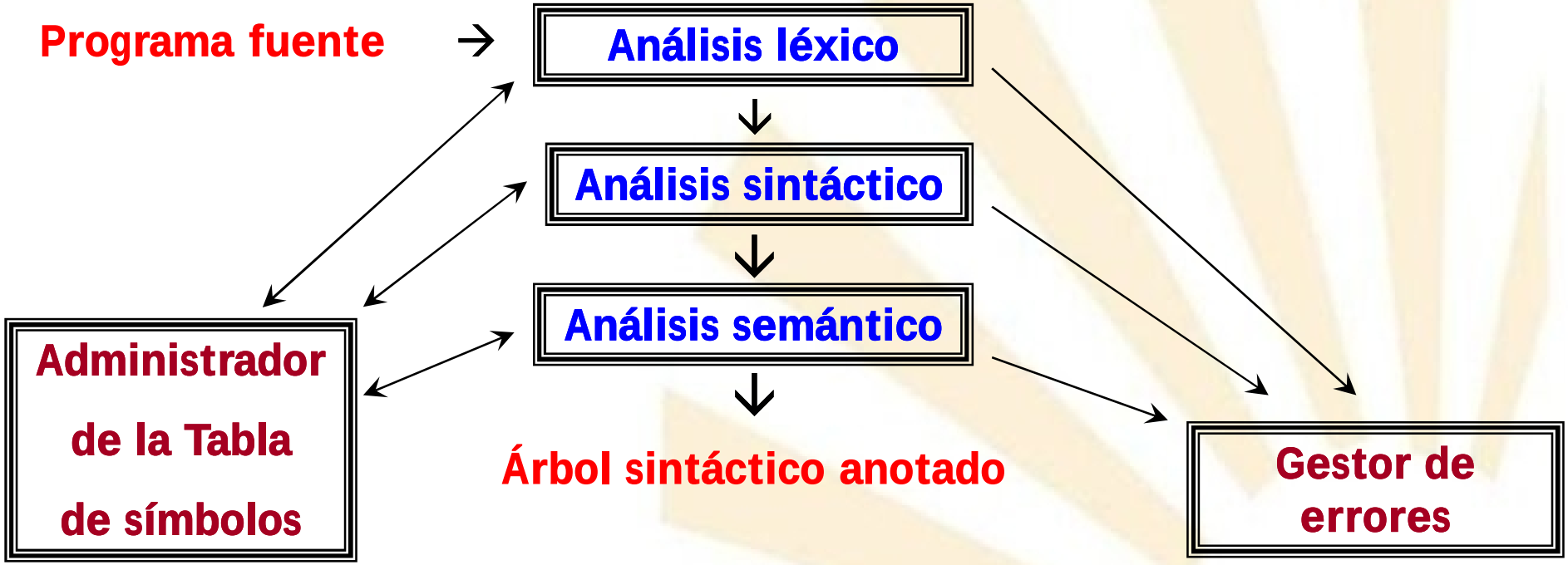
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Análisis**
      - ❑ **Análisis léxico**
      - ❑ **Análisis sintáctico**
      - ❑ **Análisis semántico**
    - ➢ **Síntesis**
      - ❑ **Generación de código intermedio**
      - ❑ **Optimización de código intermedio**
      - ❑ **Generación de código**

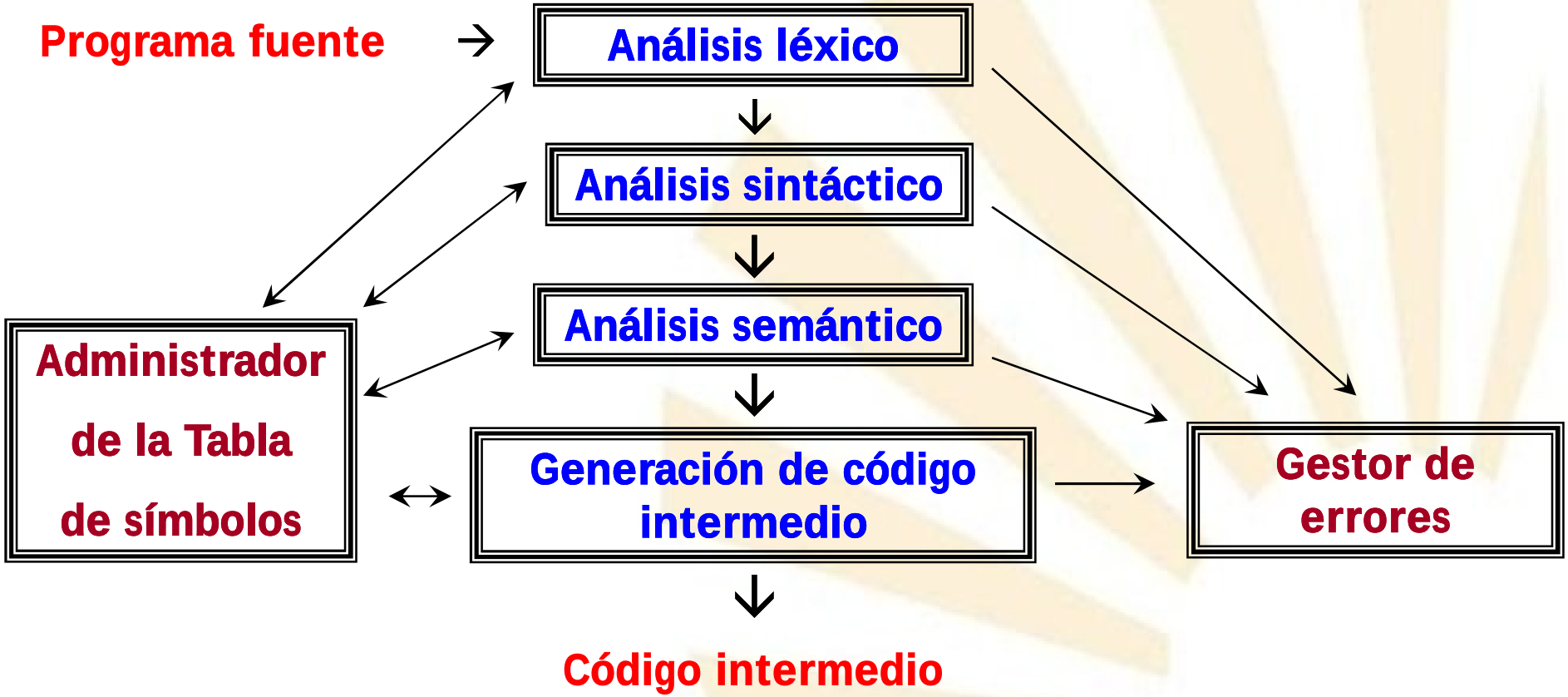
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Análisis**
      - ❑ **Análisis léxico**
      - ❑ **Análisis sintáctico**
      - ❑ **Análisis semántico**
    - ➢ **Síntesis**
      - ❑ **Generación de código intermedio**
      - ❑ **Optimización de código intermedio**
      - ❑ **Generación de código**
      - ❑ **Optimización de código**

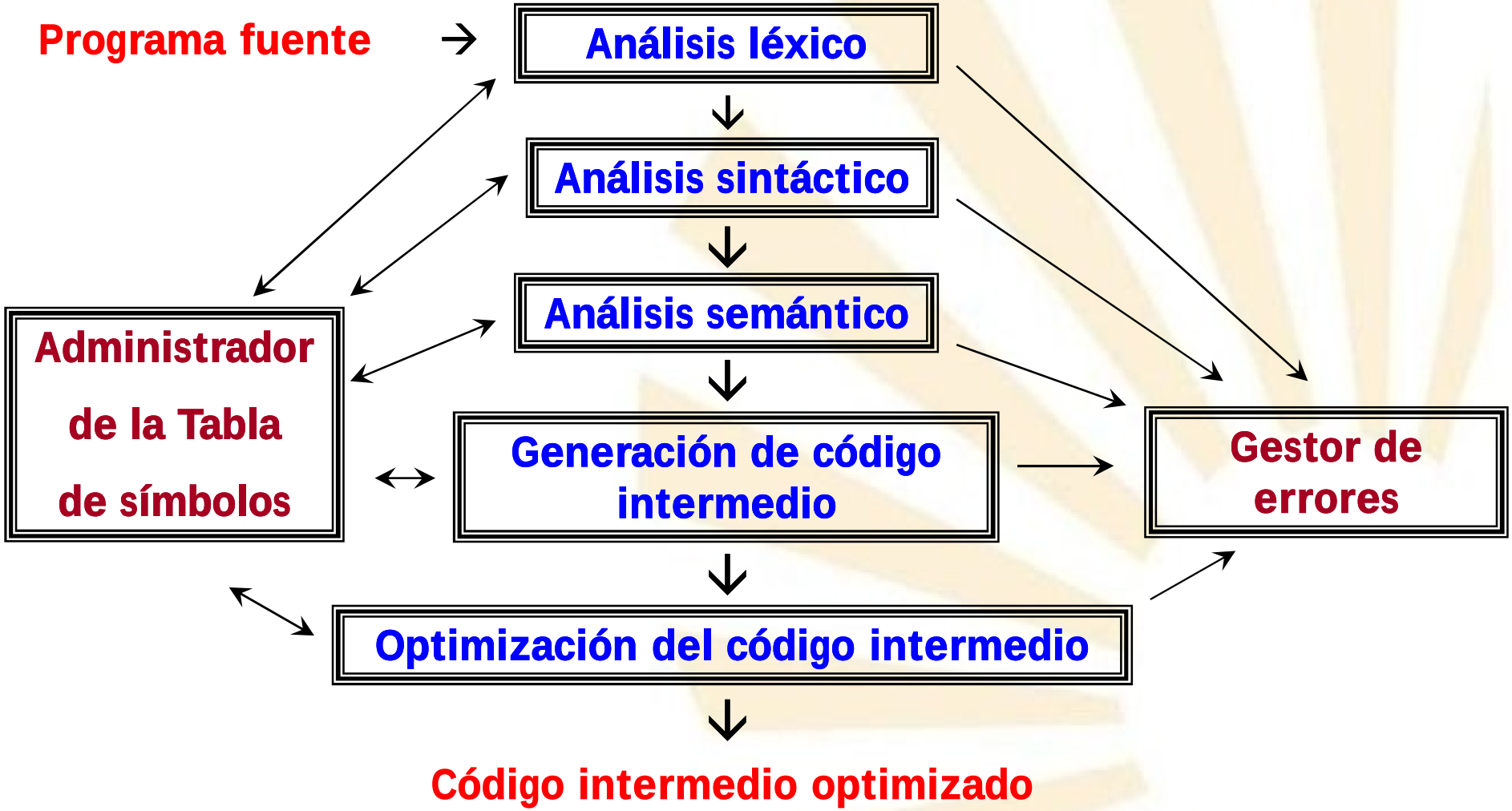
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Componentes auxiliares**
      - ❑ **Administrador de la tabla de símbolos**
      - ❑ **Gestor de errores**

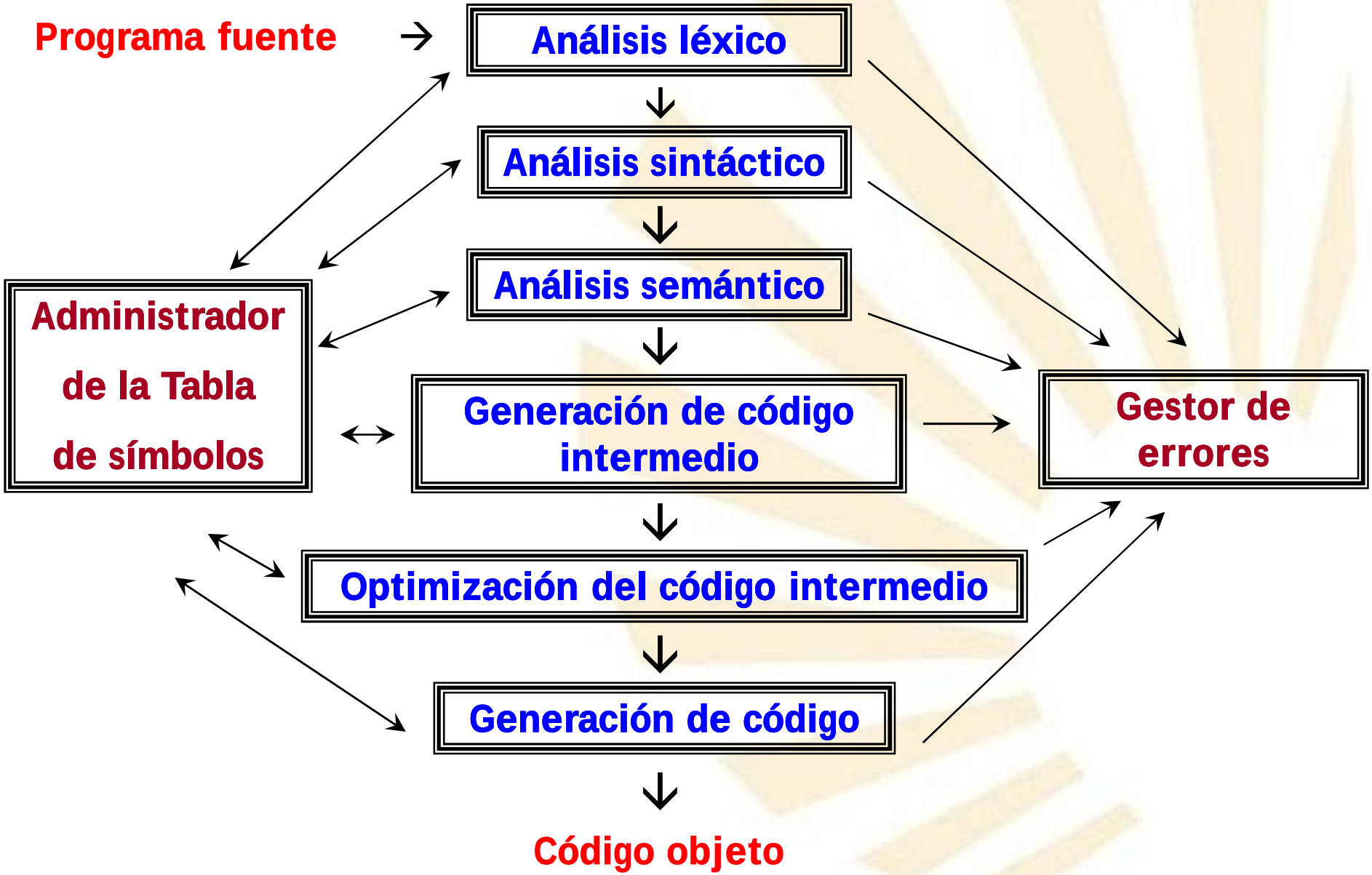

Programa fuente
Análisis léxico
Componentes léxicos
Administrador de la Tabla de símbolos
Gestor de errores

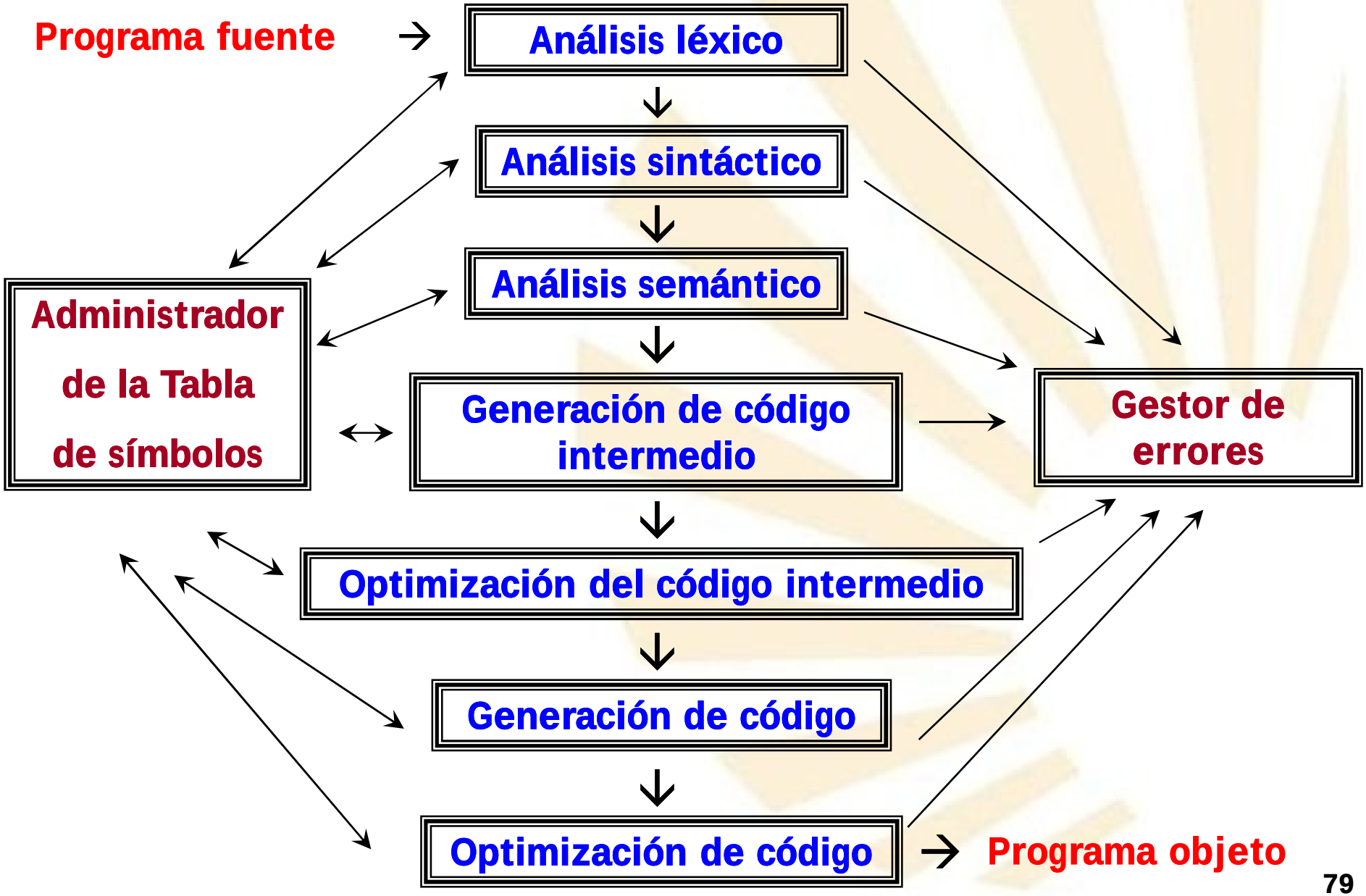

Programa fuente
Análisis léxico
Análisis sintáctico
Árbol sintáctico
Administrador de la Tabla de símbolos
Gestor de errores


Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Administrador de la Tabla de símbolos
Árbol sintáctico anotado
Gestor de errores


Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Administrador de la Tabla de símbolos
Generación de código intermedio
Gestor de errores
Código intermedio

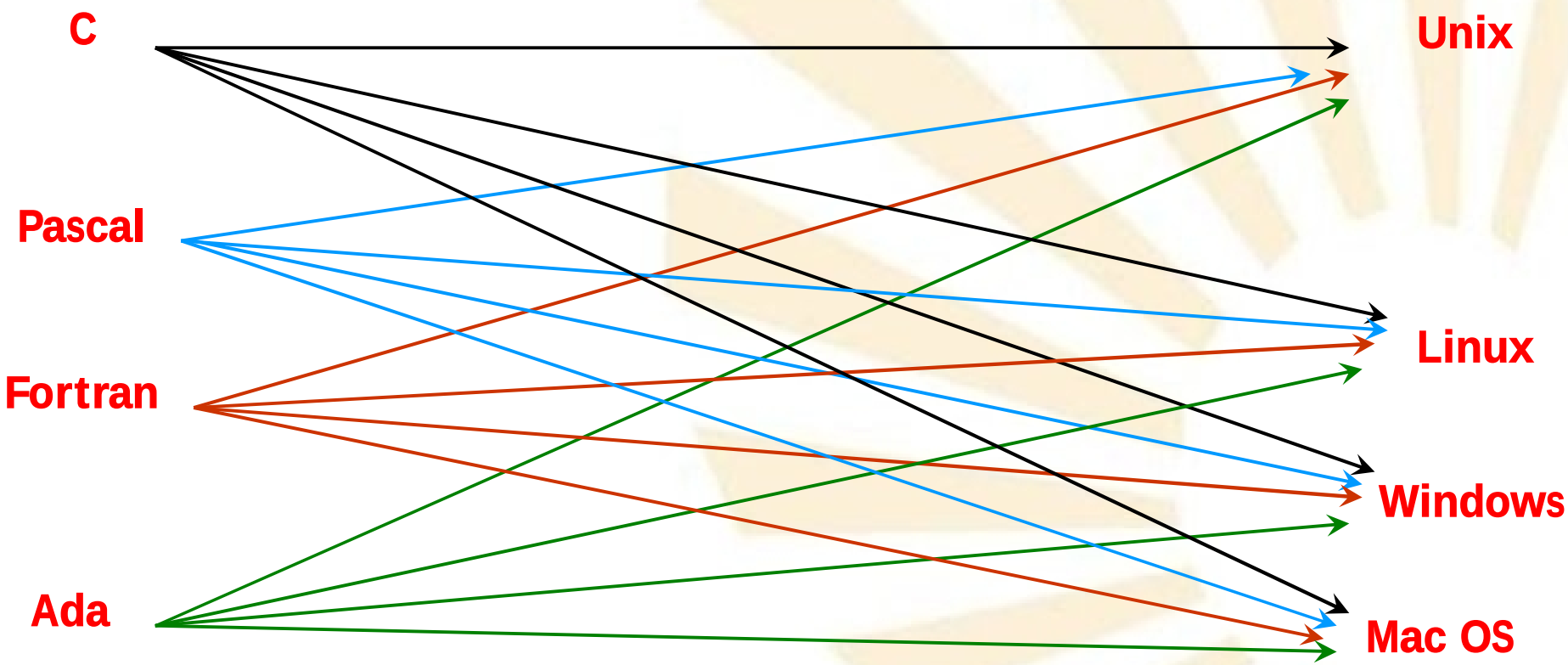

Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Administrador de la Tabla de símbolos
Generación de código intermedio
Gestor de errores
Optimización del código intermedio
Código intermedio optimizado

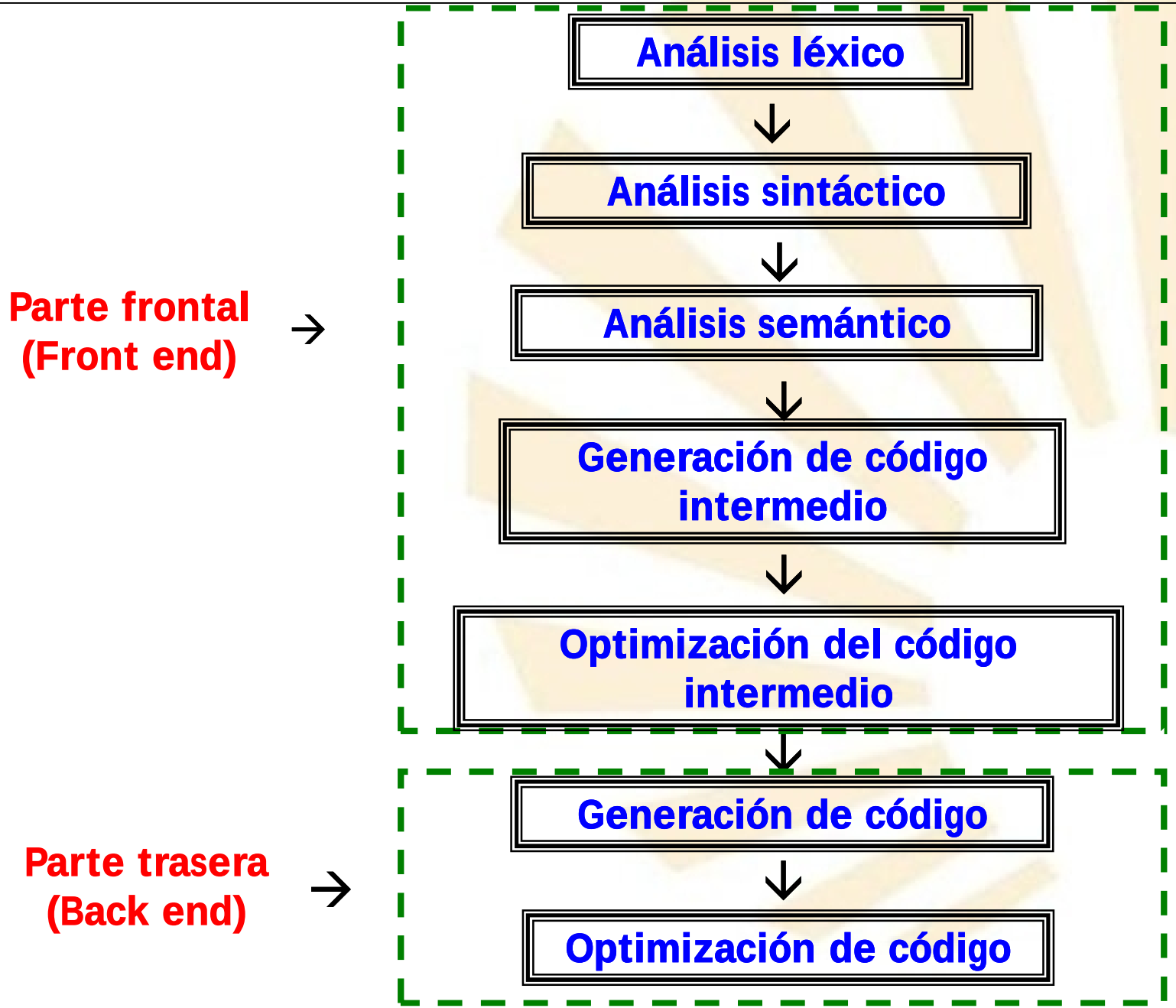

Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Administrador de la Tabla de símbolos
Generación de código intermedio
Gestor de errores
Optimización del código intermedio
Generación de código
Código objeto


Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Administrador de la Tabla de símbolos
Generación de código intermedio
Gestor de errores
Optimización del código intermedio
Generación de código
Optimización de código
Programa objeto

Necesidad de agrupar las fases de la compilación

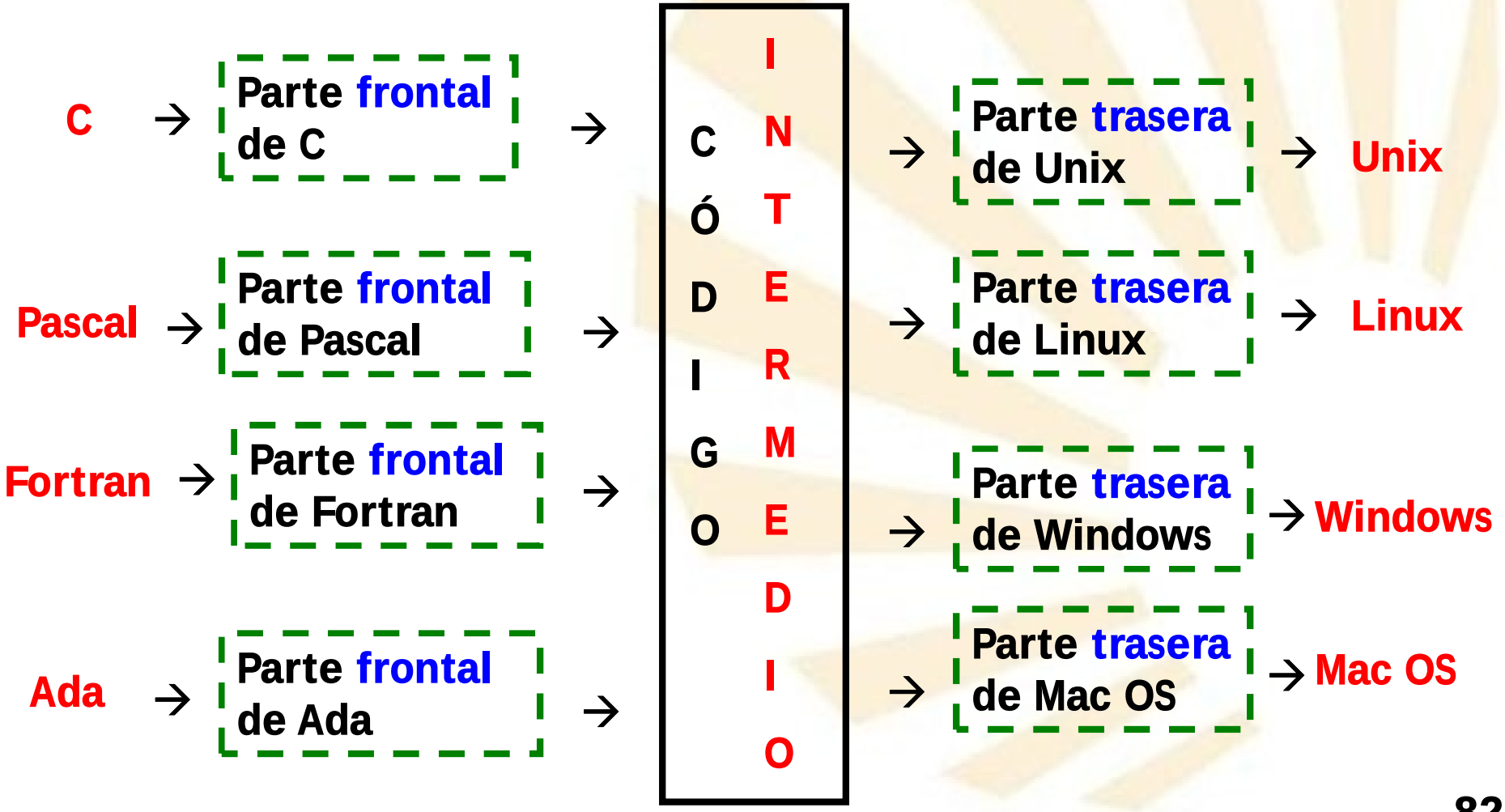
4 Lenguajes de programación × 4 Sistemas operativos = 16 compiladores




Parte frontal
(Front end)
Análisis léxico
Análisis sintáctico
Análisis semántico
Generación de código intermedio
Optimización del código intermedio
Parte trasera
(Back end)
Generación de código
Optimización de código

4 Partes frontales + 4 Partes traseras = 8 Partes

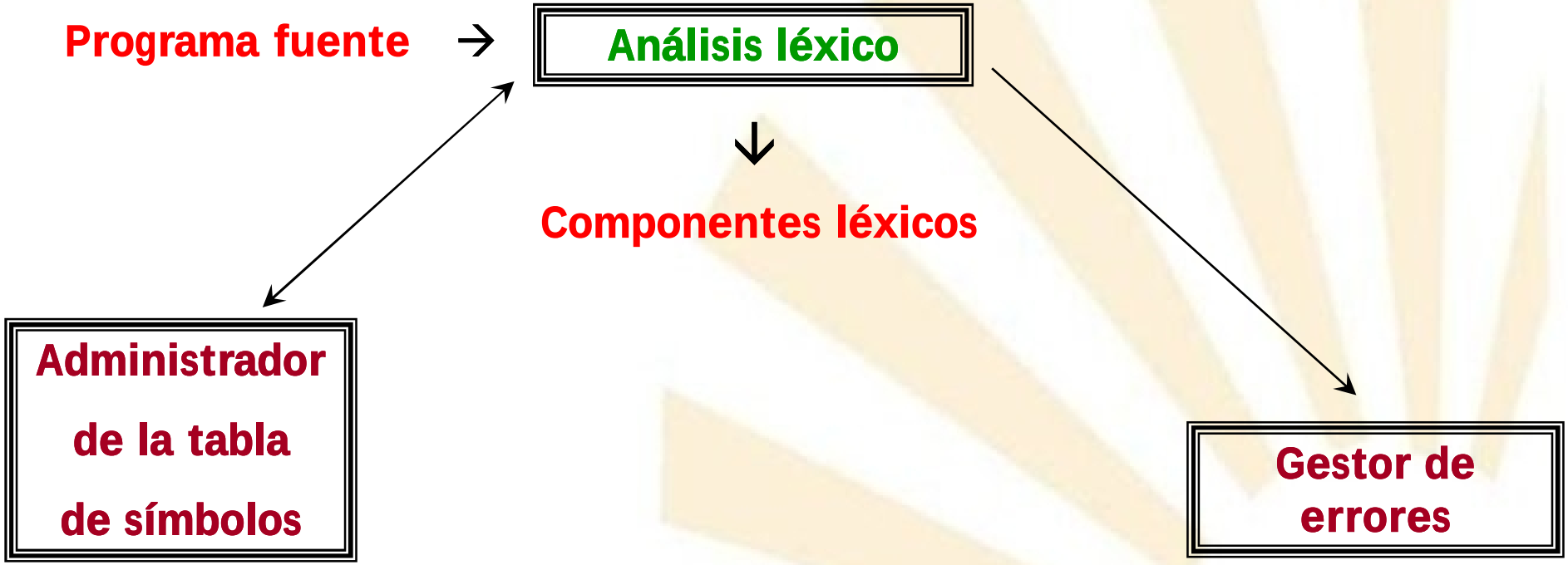
4 Partes frontales × 4 Partes traseras = 16 Compiladores



- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Análisis**
      - ❑ **Se encarga de comprobar que el programa fuente está bien escrito.**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Análisis**
      - ❑ **Análisis léxico**
      - ❑ **Análisis sintáctico**
      - ❑ **Análisis semántico**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
    - ➢ **Análisis sintáctico**
    - ➢ **Análisis semántico**


Programa fuente
Análisis léxico
Componentes léxicos
Administrador de la tabla de símbolos
Gestor de errores

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
      - ❑ También denominado análisis lexicográfico, análisis lineal, explorador o “scanner”.

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
      - ❑ También denominado análisis lexicográfico, análisis lineal, explorador o "scanner".
      - ❑ **Única fase** que tiene **contacto** con el código del **programa fuente**: favorece la modularidad y la interactividad.

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
      - ❑ También denominado análisis lexicográfico, análisis lineal, explorador o "scanner".
      - ❑ **Única fase** que tiene **contacto** con el código del **programa fuente**: favorece la modularidad y la interactividad.
      - ❑ **Objetivo**:
        - ▪ Leer el programa fuente carácter a carácter y obtener los **componentes léxicos** o "**tokens**"

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
      - ❑ También denominado análisis lexicográfico, análisis lineal, explorador o "scanner".
      - ❑ **Única fase** que tiene **contacto** con el código del **programa fuente**: favorece la modularidad y la interactividad.
      - ❑ **Objetivo**:
        - Leer el programa fuente carácter a carácter y obtener los **componentes léxicos** o "**tokens**"

Programa fuente → **Analizador léxico** → **Componentes léxicos**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
      - ❑ **Componente léxico** o "token": agrupación de caracteres con significado propio.
        - **Palabras reservadas:** if, else, while, …
        - **Identificadores:** dato, mayor, bandera, …
        - **Operadores aritméticos:** +, -, *, /, div, mod, …
        - **Operadores relacionales:** <, <=, >, >=, …
        - **Signos de puntuación:** {, }, (,), ;, …
        - **Etc.**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
      - ❑ **if** (divisor != 0.0) dividendo = divisor * cociente + resto ;
      - ❑ Componentes léxicos **enviados** al análisis sintáctico:

- **Palabra clave IF: if**
- **Los espacios en blanco son suprimidos**
- **Paréntesis izquierdo: (**
- **Identificador: divisor**
- **Operador relacional DISTINTO: !=**
- **Número: 0.0**
- **Paréntesis derecho:)**

- **Identificador**: **dividendo**
- **Símbolo de asignación**: **=**
- **Identificador**: **divisor**
- **Operador aritmético de multiplicación**: *****
- **Identificador**: **cociente**
- **Operador aritmético de adición**: **+**
- **Identificador**: **resto**
- **Delimitador de fin de sentencia**: **;**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Tabla de símbolos**

Nombre	Atributo 1	Atributo 2	...
cociente	...	...	...
dividendo	...	...	...
divisor	...	...	...
resto	...	...	...
...	...	...	...

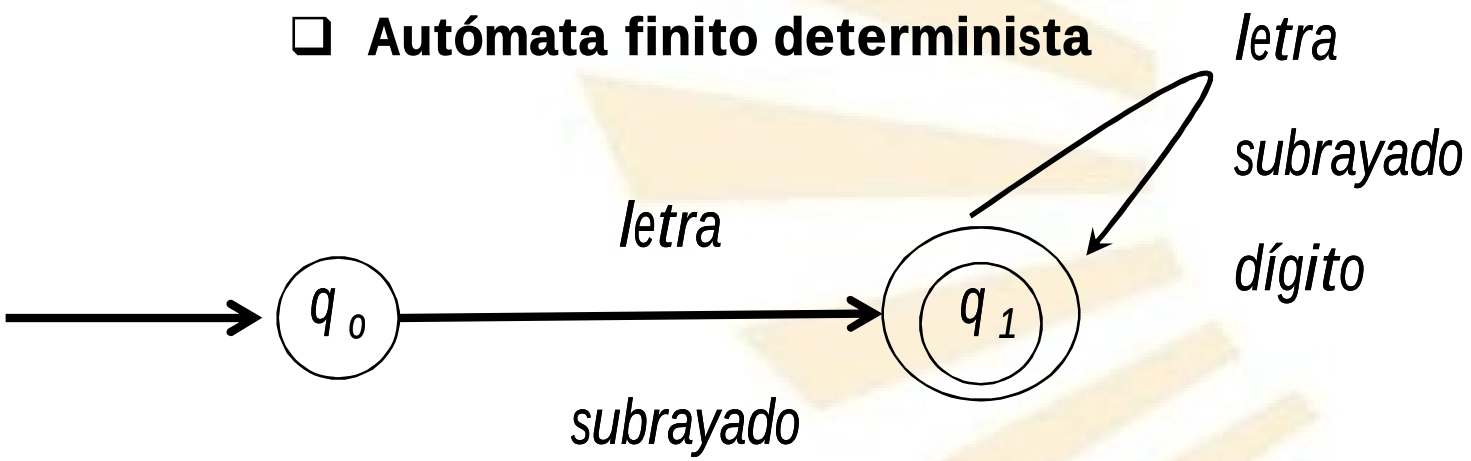
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
      - ❑ Componentes léxicos **eliminados**
        - ▪ Los **espacios en blanco**, **tabuladores** y **saltos de línea**.
        - ▪ Los **comentarios**.
      - ❑ Estos componentes léxicos
        - ▪ Favorecen la lectura y comprensión de los programas.
        - ▪ Pero **no son necesarios** para generar el código ejecutable.
      - ❑ Generalmente, el análisis **léxico** es una subrutina o procedimiento **auxiliar** del análisis **sintáctico**.

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
      - ❑ Se utilizan las **expresiones regulares** para definir las características de los componentes léxicos.
      - ❑ A partir de las expresiones regulares, se **genera** el **analizador léxico** que **simula** el funcionamiento de un **autómata finito determinista**.
      - ❑ Nota
        - ▪ Véase el tema nº 2

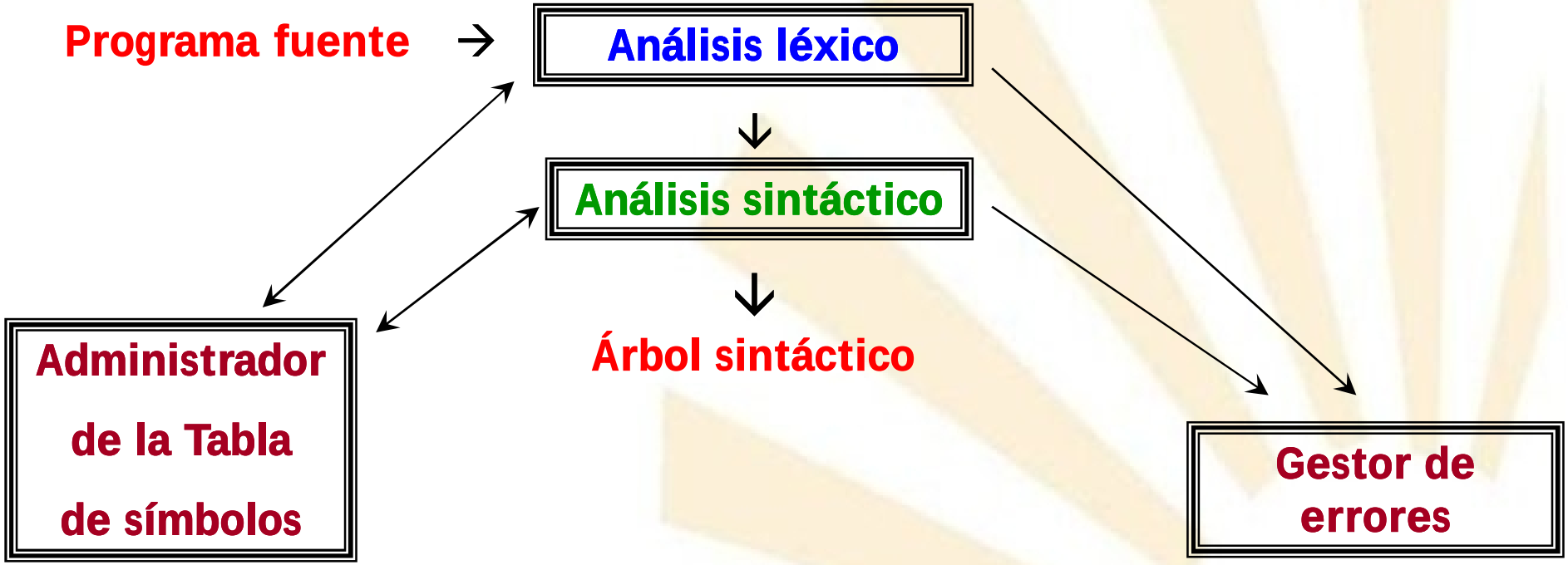
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
      - ❑ Ejemplo:
        - ▪ **Expresión regular** que denota los identificadores del lenguaje C

          *(letra + subrayado) (letra + subrayado + dígito)**

        - ❑ **Autómata finito determinista**

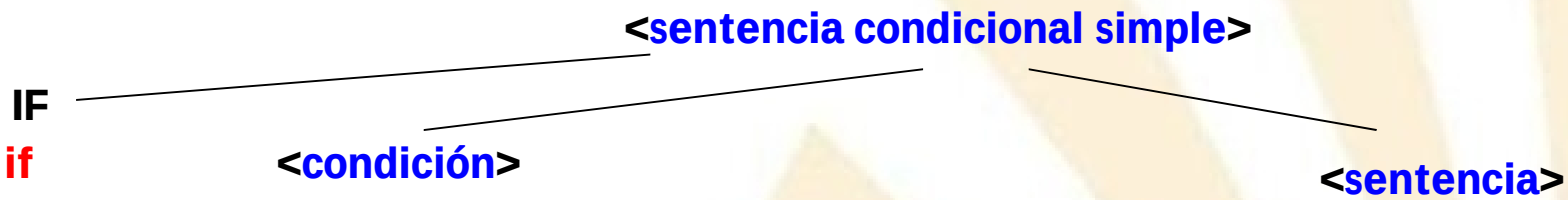


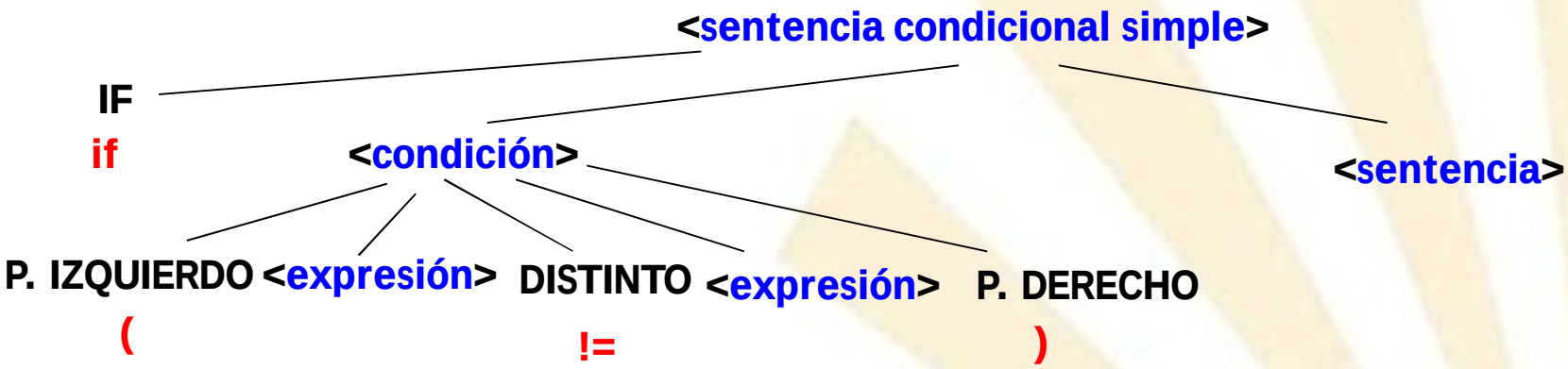
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
    - ➢ **Análisis sintáctico**
    - ➢ **Análisis semántico**

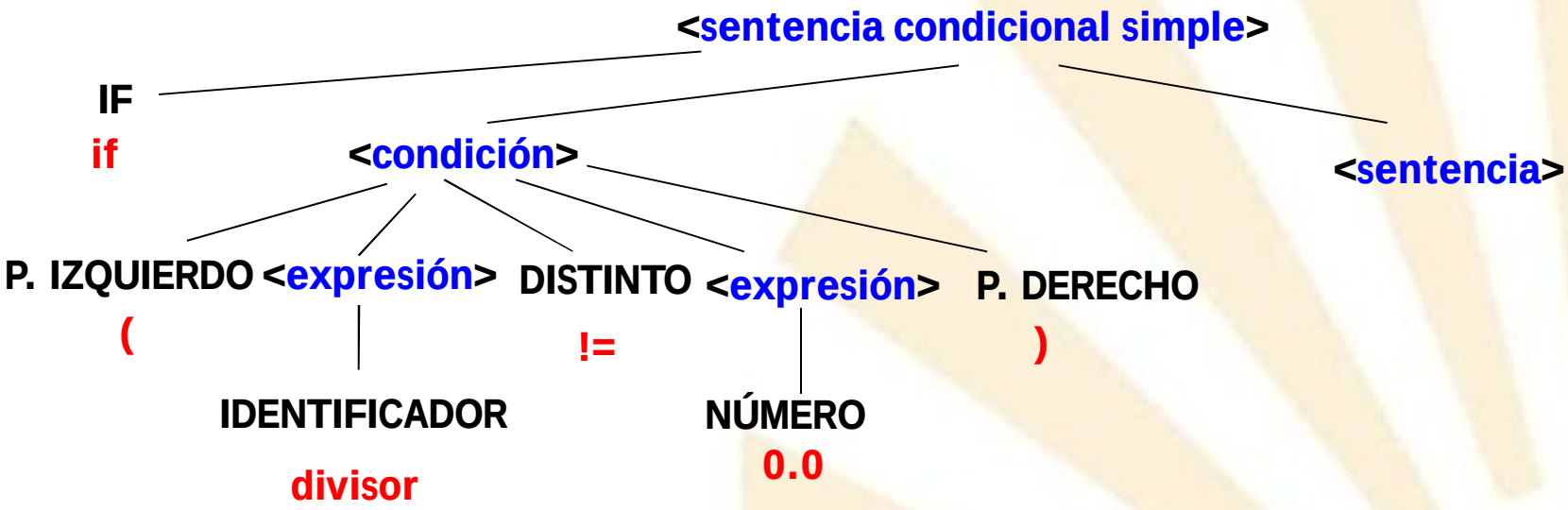

Programa fuente
Análisis léxico
Análisis sintáctico
Árbol sintáctico
Administrador de la Tabla de símbolos
Gestor de errores

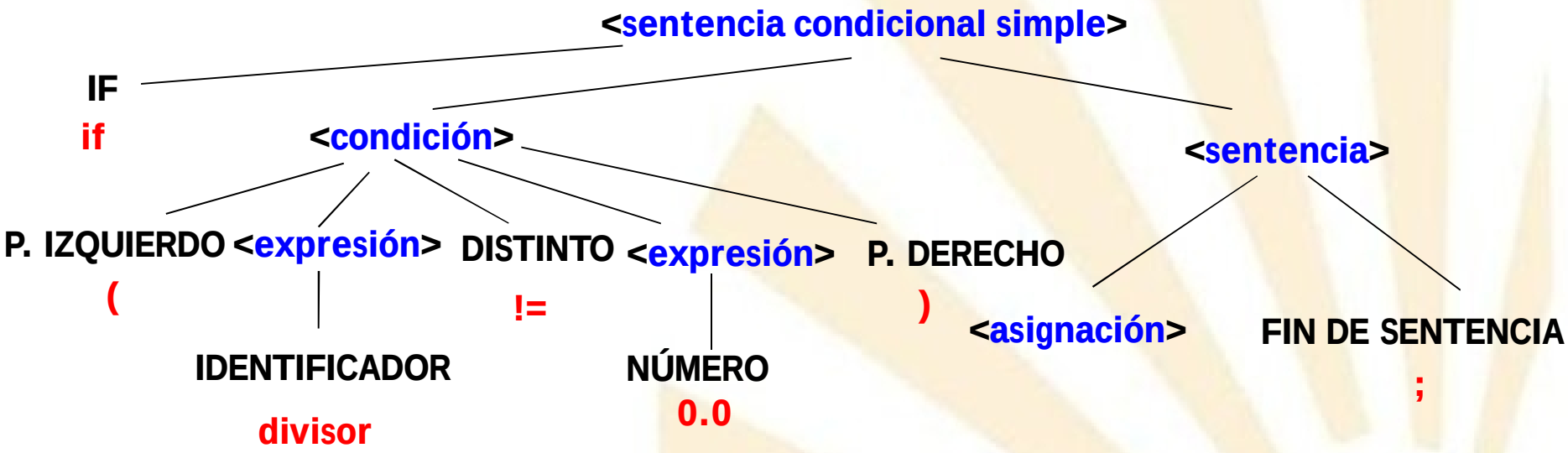
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis sintáctico**
      - ❑ También denominado análisis jerárquico o gramatical o "parser".
      - ❑ **Objetivos**:
        - ▪ Comprobar **la sintaxis del código fuente**: utiliza las **reglas gramaticales** del lenguaje fuente y los componentes léxicos.
        - ▪ Generar una **representación jerárquica** (figurada): **árbol sintáctico.**

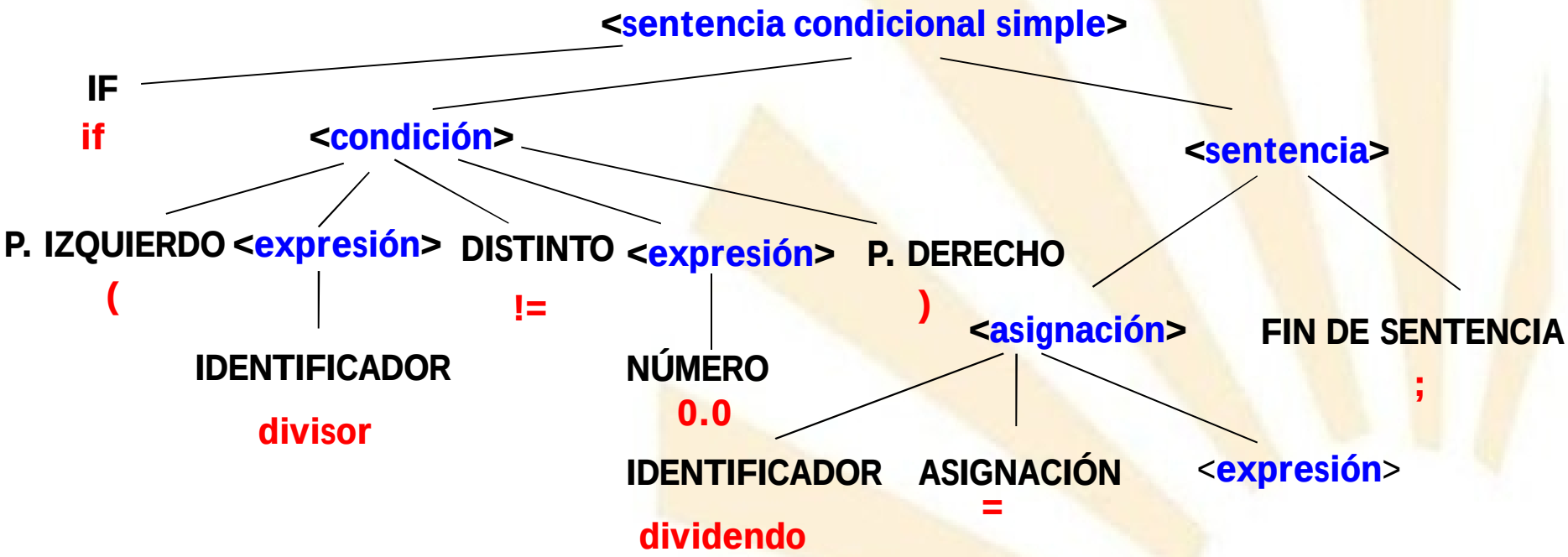
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis sintáctico**
      - ❑ Se utilizan **gramáticas de contexto libre** para definir la **sintaxis de las sentencias** de los lenguajes de programación.
      - ❑ A partir de la gramática de contexto libre, se **genera** el **analizador sintáctico** que **simula** el funcionamiento de un **autómata con pila**.
      - ❑ Nota
        - ▪ Véanse los temas nº 3, 4 y 5.

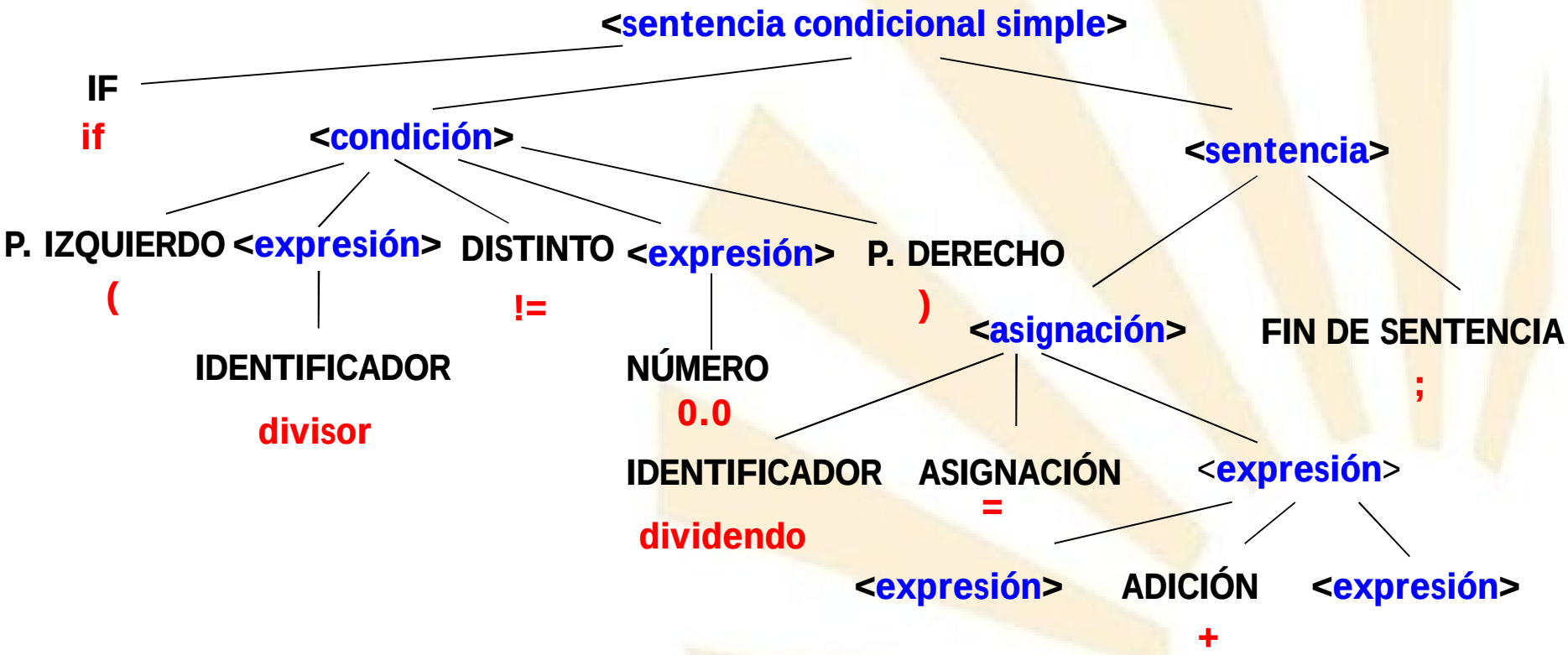

<sentencia condicional simple>
IF
if
<condición>
<sentencia>

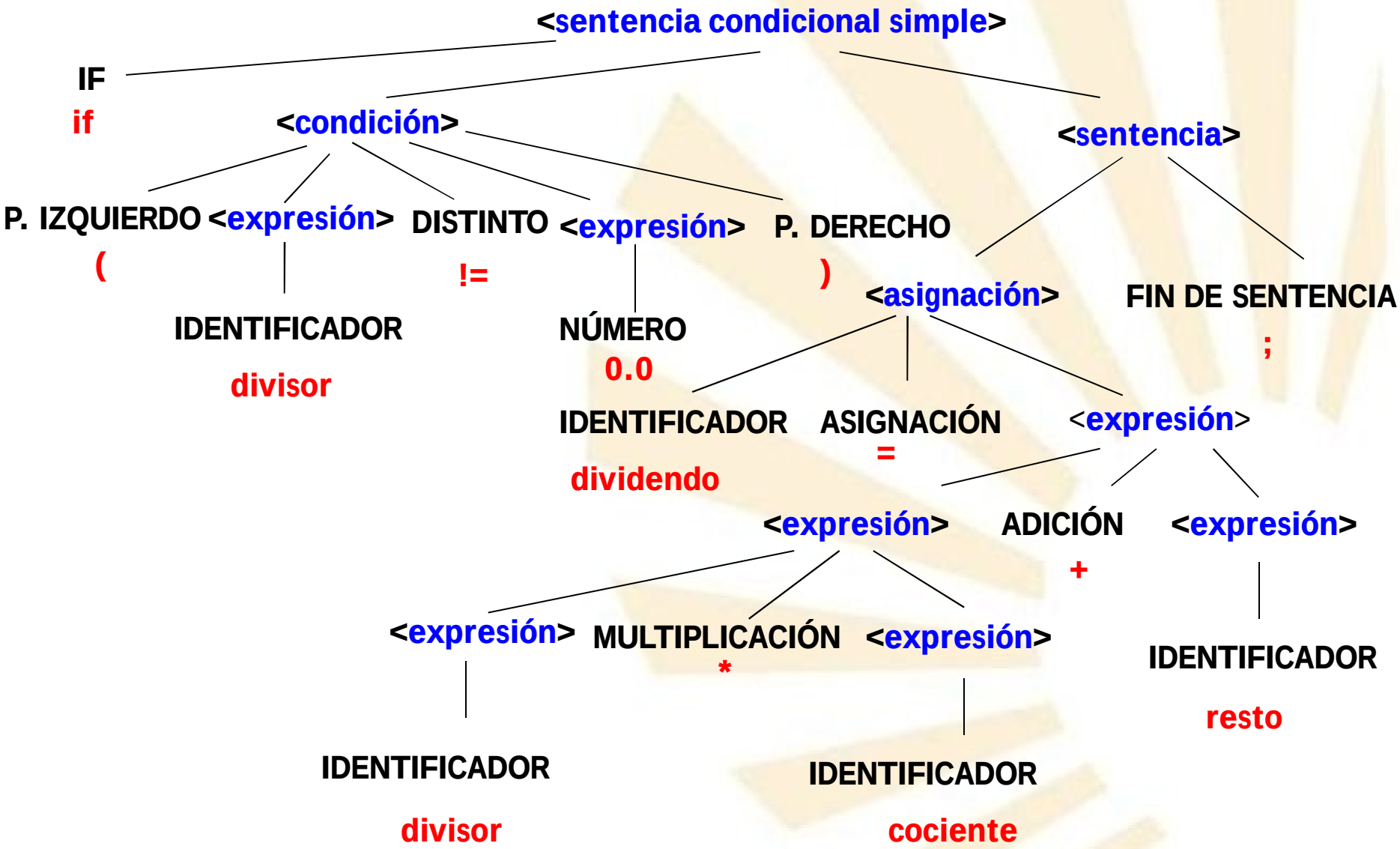

<sentencia condicional simple>
IF
if
<condición>
<sentencia>
P. IZQUIERDO
(
<expresión>
DISTINTO
!=
<expresión>
P. DERECHO
)

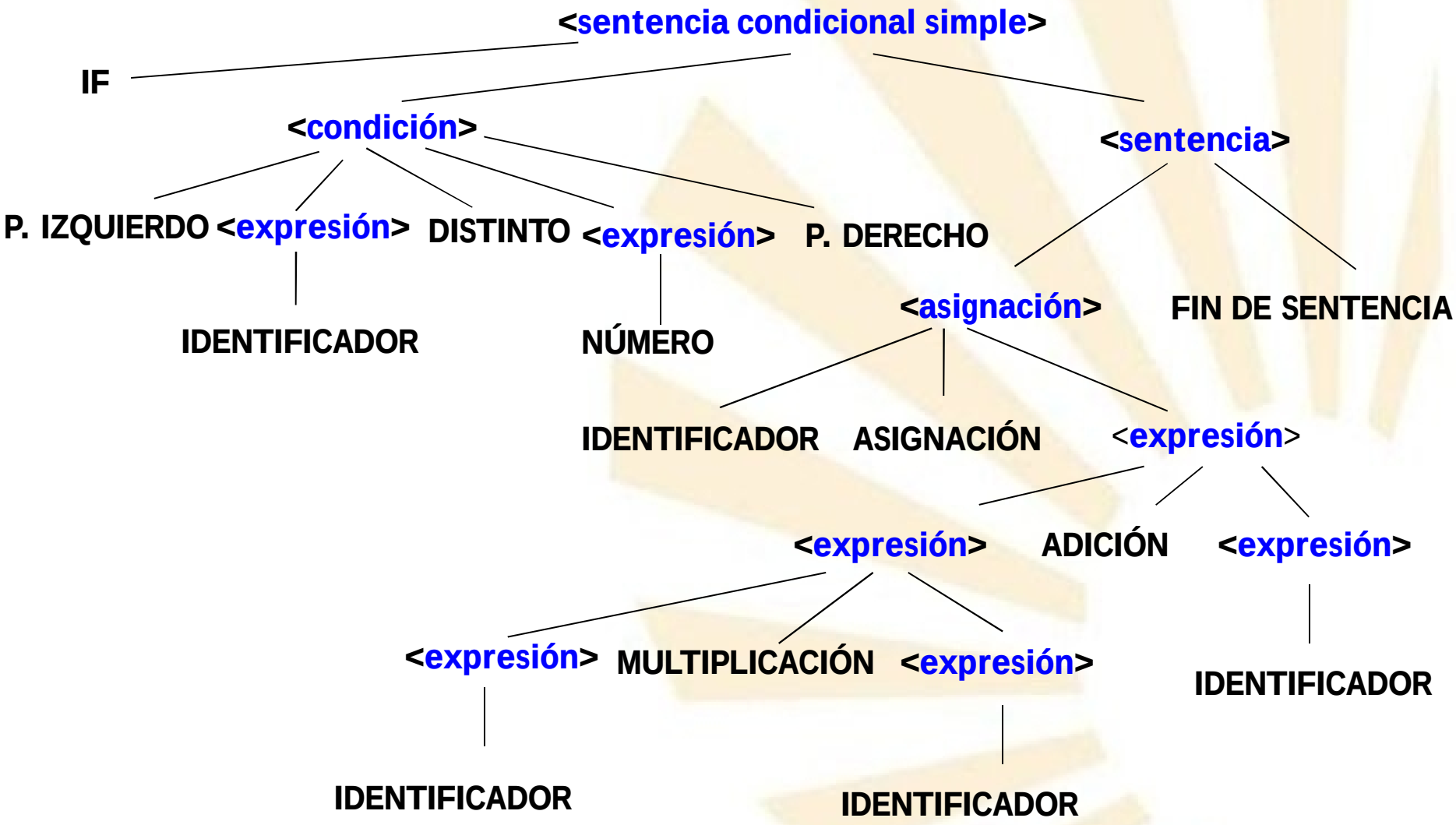

<sentencia condicional simple>
IF
if
<condición>
<sentencia>
P. IZQUIERDO
(
<expresión>
IDENTIFICADOR
divisor
DISTINTO
!=
<expresión>
NÚMERO
0.0
P. DERECHO
)


<sentencia condicional simple>
IF
if
<condición>
<sentencia>
P. IZQUIERDO
(
<expresión>
IDENTIFICADOR
divisor
DISTINTO
!=
<expresión>
NÚMERO
0.0
P. DERECHO
)
<asignación>
FIN DE SENTENCIA
;

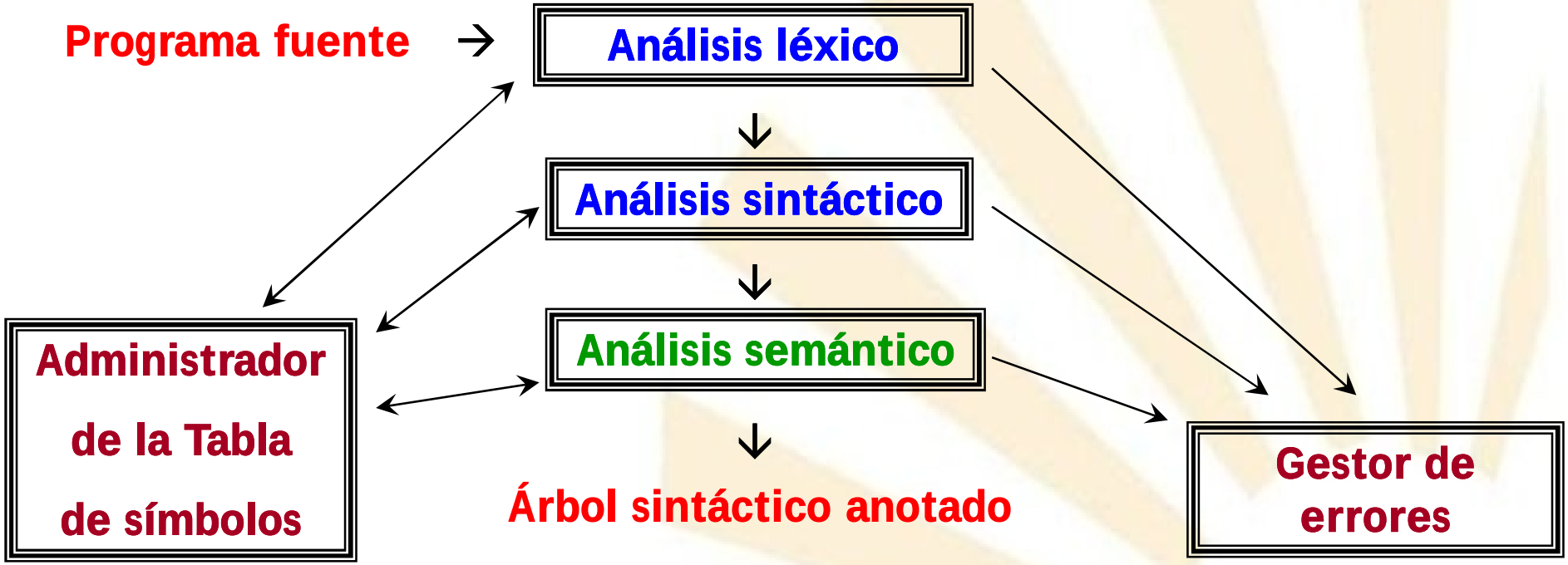

<sentencia condicional simple>
IF
if
<condición>
<sentencia>
P. IZQUIERDO
(
<expresión>
IDENTIFICADOR
divisor
DISTINTO
!=
<expresión>
NÚMERO
0.0
P. DERECHO
)
<asignación>
FIN DE SENTENCIA
;
IDENTIFICADOR
dividendo
ASIGNACIÓN
=
<expresión>


<sentencia condicional simple>
IF
if
<condición>
<sentencia>
P. IZQUIERDO
(
<expresión>
IDENTIFICADOR
divisor
DISTINTO
!=
<expresión>
NÚMERO
0.0
P. DERECHO
)
<asignación>
FIN DE SENTENCIA
;
IDENTIFICADOR
dividendo
ASIGNACIÓN
=
<expresión>
<expresión>
ADICIÓN
+
<expresión>


<sentencia condicional simple>
IF
if
<condición>
<sentencia>
P. IZQUIERDO
(
<expresión>
DISTINTO
!=
<expresión>
P. DERECHO
)
<asignación>
FIN DE SENTENCIA
;
IDENTIFICADOR
divisor
NÚMERO
0.0
IDENTIFICADOR
dividendo
ASIGNACIÓN
=
<expresión>
<expresión>
ADICIÓN
+
<expresión>
<expresión>
MULTIPLICACIÓN
*
<expresión>
IDENTIFICADOR
resto
IDENTIFICADOR
divisor
IDENTIFICADOR
cociente


<sentencia condicional simple>
IF
<condición>
<sentencia>
P. IZQUIERDO
<expresión>
DISTINTO
<expresión>
P. DERECHO
<asignación>
FIN DE SENTENCIA
IDENTIFICADOR
NÚMERO
IDENTIFICADOR
ASIGNACIÓN
<expresión>
<expresión>
ADICIÓN
<expresión>
<expresión>
MULTIPLICACIÓN
<expresión>
IDENTIFICADOR
IDENTIFICADOR
IDENTIFICADOR

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis léxico**
    - ➢ **Análisis sintáctico**
    - ➢ **Análisis semántico**

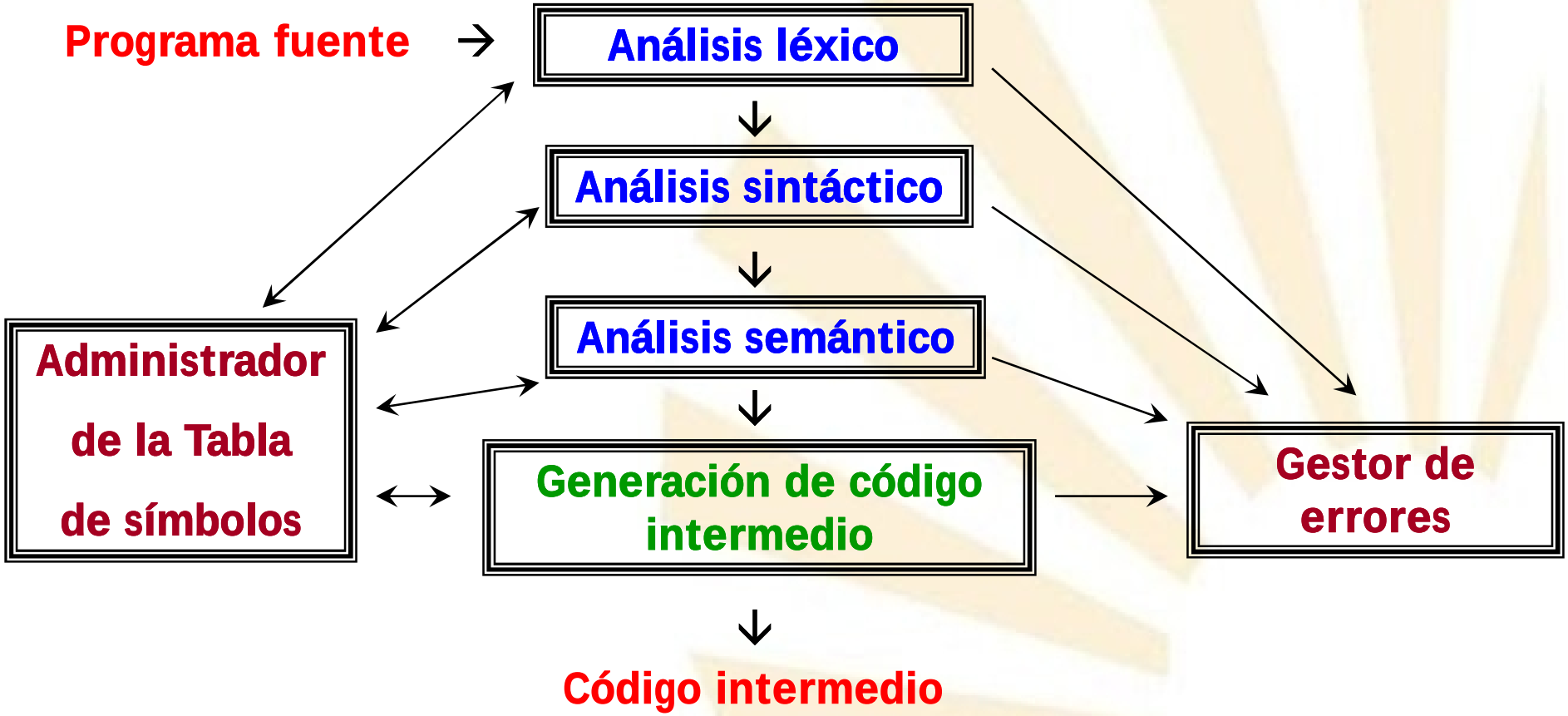

Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Administrador de la Tabla de símbolos
Árbol sintáctico anotado
Gestor de errores

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Análisis**
    - ➢ **Análisis semántico**
      - ❑ **Comprueba si el significado de las sentencias es correcto**.
      - ❑ Utiliza el árbol sintáctico y la tabla de símbolos.
      - ❑ Algunos de los errores semánticos que pueden detectar:
        - Operandos y operadores incompatibles.
        - Diferencia de tipos entre los argumentos reales y los argumentos formales.
        - Etc.
      - ❑El análisis **semántico** suele estar **integrado** en el análisis **sintáctico**.

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Análisis**
    - ➢ **Síntesis**
      - ❑ Se encarga de **transformar** la **representación** obtenida durante el **análisis** en el **código objeto o ejecutable**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Generación de código intermedio**
    - ➢ **Optimización del código intermedio**
    - ➢ **Generación de código**
    - ➢ **Optimización del código**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Generación de código intermedio**
    - ➢ **Optimización del código intermedio**
    - ➢ **Generación de código**
    - ➢ **Optimización del código**


Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Generación de código intermedio
Administrador de la Tabla de símbolos
Gestor de errores
Código intermedio

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Generación de código intermedio**
      - ❑ **Genera** una **representación intermedia** del código fuente que ha de tener las siguientes características
        - ▪ Ha de ser fácil de generar a partir del código fuente.
        - ▪ Ha de ser fácil de traducir al código objeto o ejecutable

Código fuente → **Código intermedio** → **Código objeto o ejecutable**

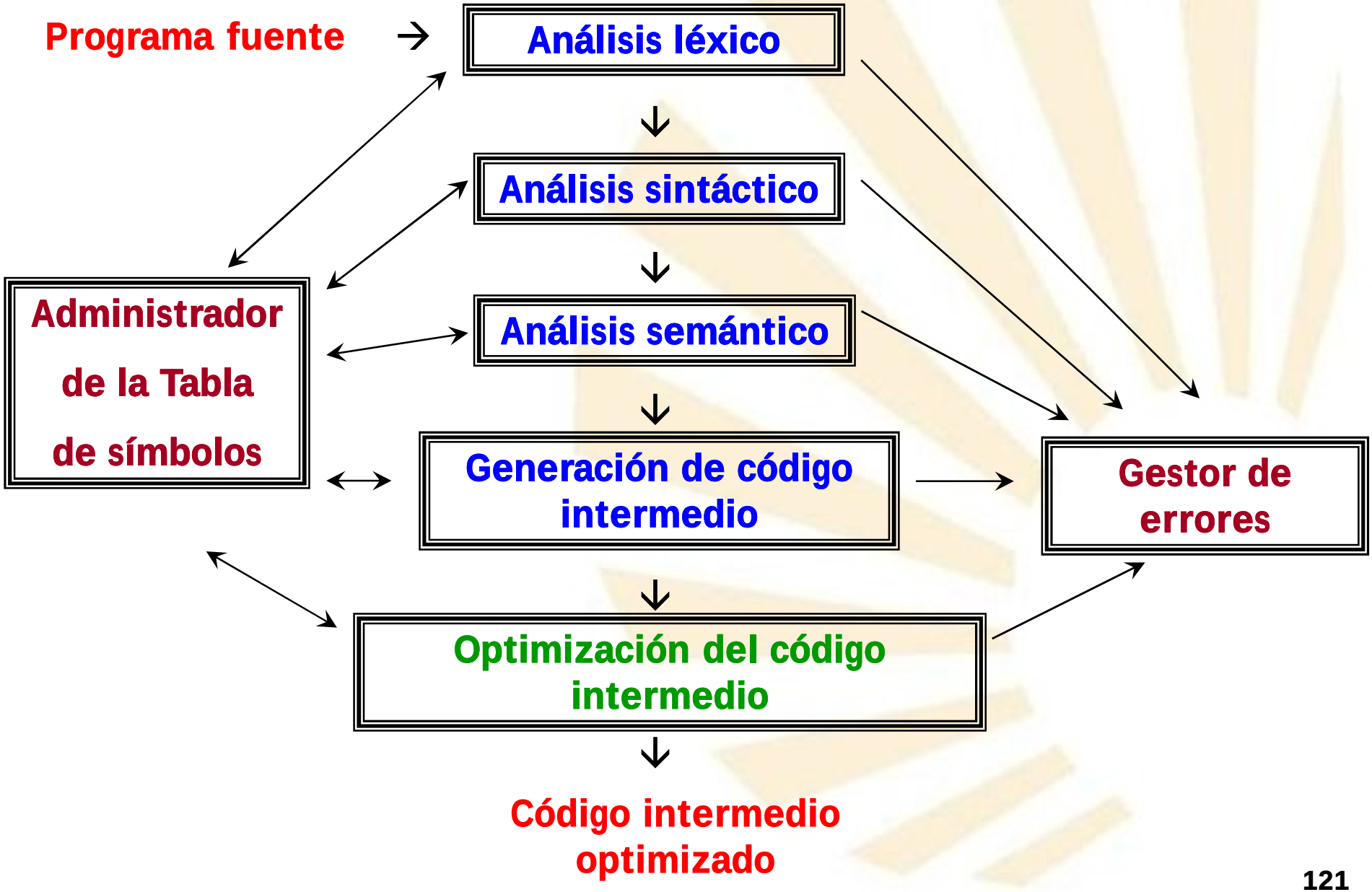
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Generación de código intermedio**
      - ❑ Genera un **representación intermedia** del código fuente
      - ❑ Es una **fase** opcional, pero **muy recomendable**.
        - ▪ "**Redestinación**" :
          - ❖ Al integrarse en la "**parte frontal**" del compilador, favorece la generación de código objeto para **distintos entornos de ejecución**.
        - ▪ **Optimización independiente del entorno de ejecución**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Generación de código intermedio**
      - ❑ Se utilizan **definiciones dirigidas por la sintaxis** o **esquemas de traducción** que se incorporan al análisis sintáctico.
      - ❑ Tipos de representaciones intermedias:
        - Notación postfija
        - Árboles sintácticos
        - Grafos dirigidos acíclicos
        - **Código de tres direcciones**:
          - ❖ Cuádruplas, triples y triples indirectos.

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Generación de código intermedio**
      - ❑ **if** (divisor != 0.0) dividendo = divisor * cociente + resto ;
      - ❑ Representación intermedia en código de tres direcciones:

```
100. if divisor = 0 goto 104
101. t1 := divisor * cociente
102. t2 := t1 + resto
103. dividendo := t2
104. …
```

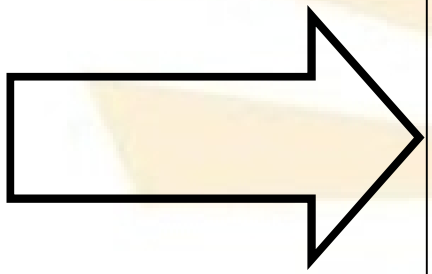
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Generación de código intermedio**
    - ➢ **Optimización del código intermedio**
    - ➢ **Generación de código**
    - ➢ **Optimización del código**


Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Generación de código intermedio
Optimización del código intermedio
Código intermedio optimizado
Administrador de la Tabla de símbolos
Gestor de errores

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Optimización del código intermedio**
      - ❑ Esta fase es opcional, pero también es **recomendable**
      - ❑ **Objetivo:**
        - ▪ Realizar una **optimización** del código intermedio que sea **independiente** de la **máquina** en la que se ejecute el código objeto.
      - ❑ La optimización es un **problema NP-Completo**

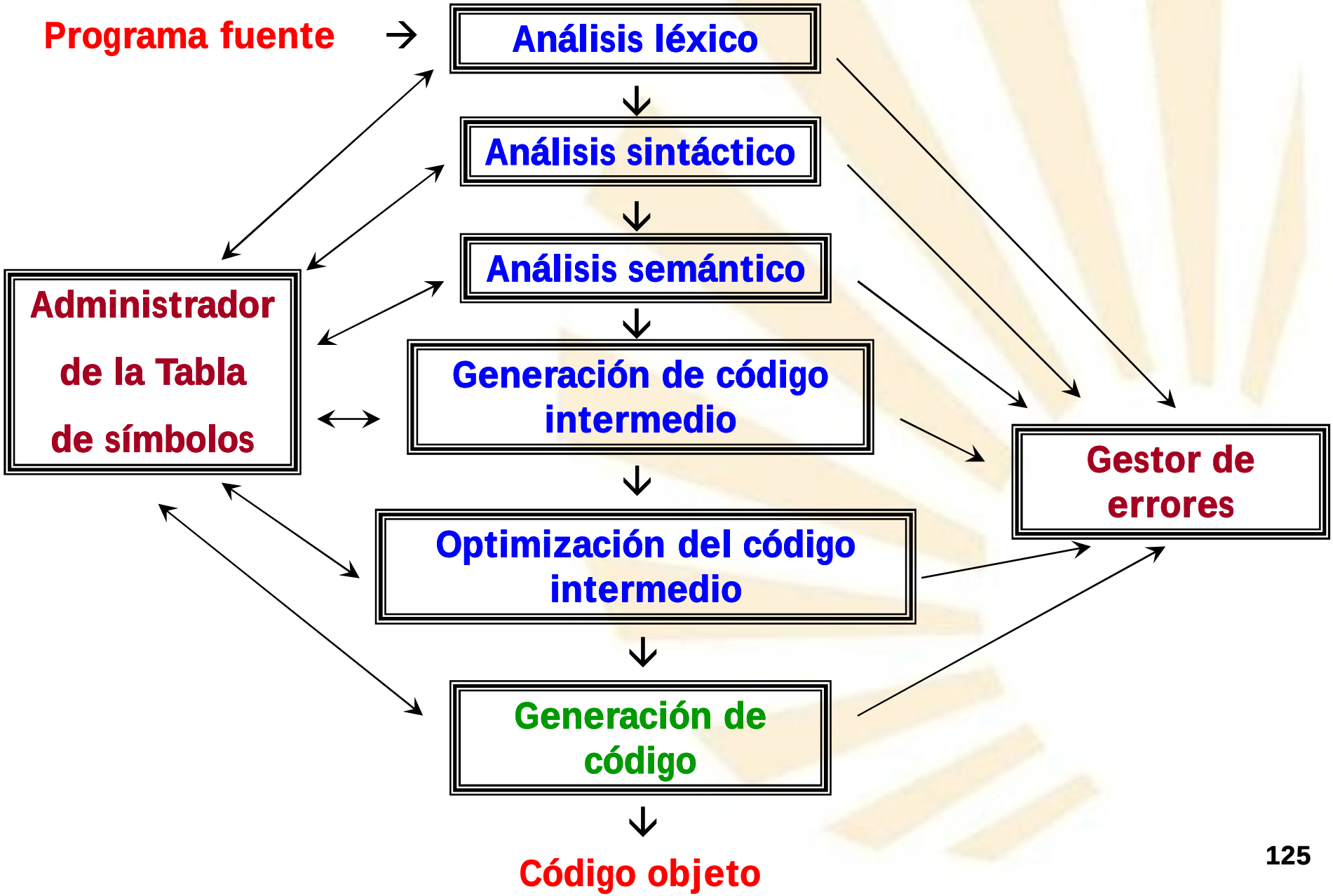
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Optimización del código intermedio**
      - ❑ **if** (divisor != 0.0) dividendo = divisor * cociente + resto ;
      - ❑ Optimización de la representación intermedia en código de tres direcciones:

```
100. if divisor = 0 goto 104
101. t1 := divisor * cociente
102. t2 := t1 + resto
103. dividendo := t2
104. …
```



```
100. if divisor = 0 goto 103
101. t1 := divisor * cociente
102. dividendo := t1 + resto
103. …
```

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Generación de código intermedio**
    - ➢ **Optimización del código intermedio**
    - ➢ **Generación de código**
    - ➢ **Optimización del código**

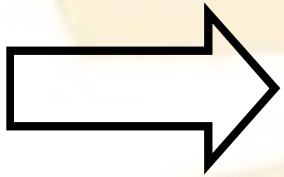

Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Generación de código intermedio
Optimización del código intermedio
Generación de código
Código objeto
Administrador de la Tabla de símbolos
Gestor de errores

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Generación de código**
      - ❑ **Objetivo**
        - Traducir la representación intermedia a código objeto o ejecutable (código máquina).
      - ❑ **Ejemplo:**
        - Se va a generar código objeto en **ensamblador**
        - Las operaciones aritmético – lógicas se han de realizar sobre **registros de máquina**: R1, R2, …
        - Las proposiciones **condicionales** son generadas mediante comparaciones (CMP) y saltos condicionales (JE, JLE, …).

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Generación de código**

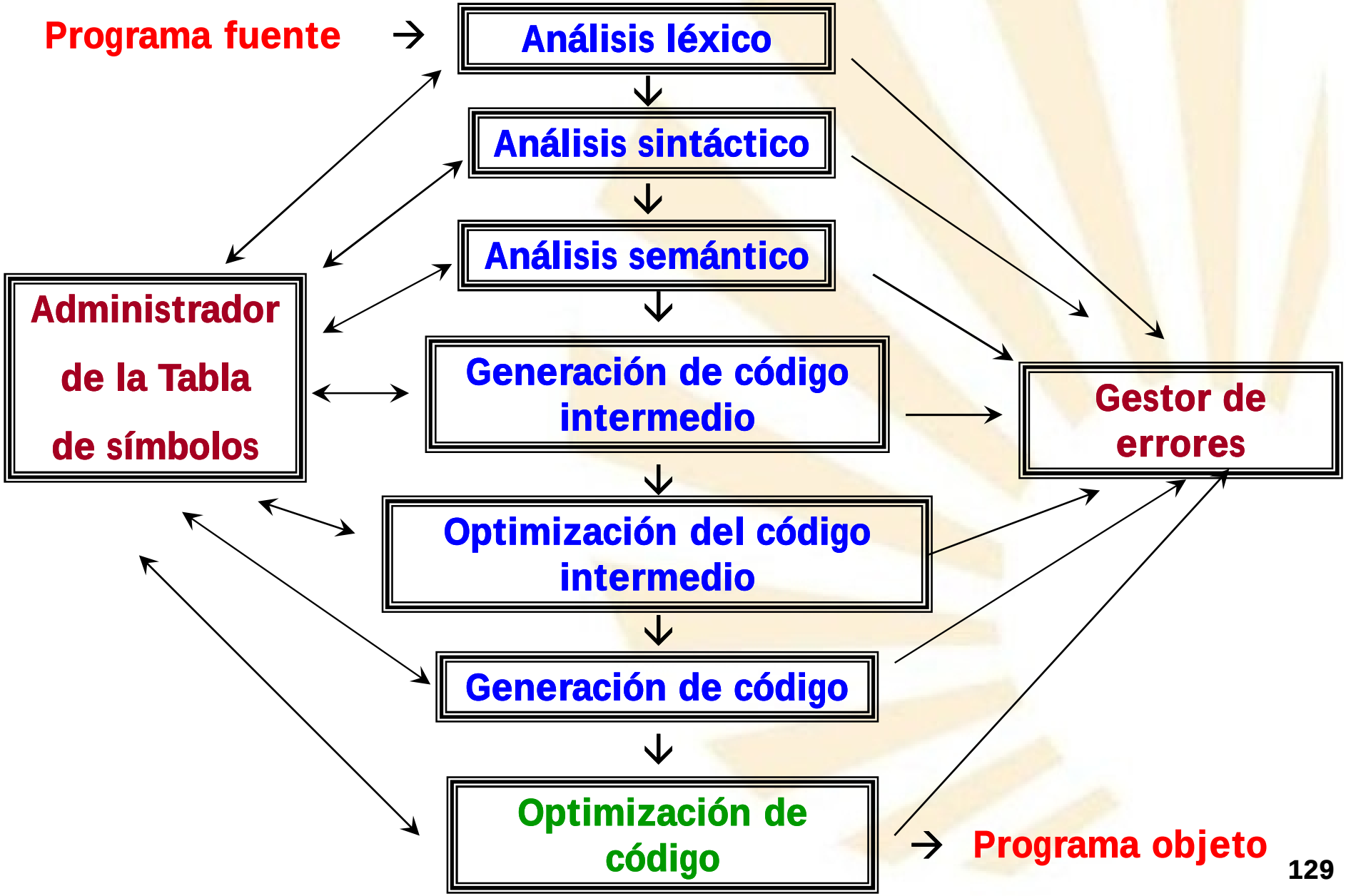
```
if (divisor != 0.0)
 dividendo = divisor * cociente + resto ;
```

```
100. if divisor = 0 goto 103
101. t1 := divisor * cociente
102. dividendo := t1 + resto
103. ...
```



```
100. MOV divisor, R1
101. CMP #0, R1
102. JE 110
103. MOV cociente, R2
104. MUL R2, R1
105. MOV R1, t1
106. MOV t1, R3
107. MOV resto, R4
108. SUM R3, R4
109. MOV R4, dividendo
110.  ...
```

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Generación de código intermedio**
    - ➢ **Optimización del código intermedio**
    - ➢ **Generación de código**
    - ➢ **Optimización del código**


Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Administrador de la Tabla de símbolos
Generación de código intermedio
Gestor de errores
Optimización del código intermedio
Generación de código
Optimización de código
Programa objeto

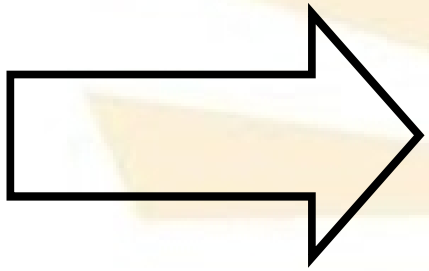
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Optimización del código**
      - ❑ Generar un código más eficiente:
        - **Ejecución** más **rápida**
        - Ocupar **menos** espacio de **memoria**.
      - ❑La optimización es un problema **NP-Completo**
      - ❑Optimizar las necesidades de tiempo y memoria de forma conjunta suele ser **difícil**
      - ❑ **Tiempo y memoria** son dos factores **contrapuestos**.
      - ❑ La optimización absoluta no siempre se puede alcanzar: sólo se producen **mejoras**, pero no se tiene garantía de que sean óptimas.

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Optimización del código**
      - ❑ Las mejores **transformaciones** son las que obtienen el **mayor beneficio** con el **menor esfuerzo**
      - ❑ Criterios:
        - ▪ Se ha de **preservar el significado** del programa
        - ▪ Debe **acelerar** los programas de forma **apreciable**
        - ▪ Tiene que **merecer la pena**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Optimización del código**
      - ❑ Posibles mejoras del código:
        - No evaluación repetida de expresiones comunes
        - Evitar la propagación de copias
        - Supresión de código inactivo o "muerto": análisis de control de flujo.
        - Optimización de bucles: no evaluación de expresiones constantes dentro de los bucles
        - Reutilización de registros de máquina.
        - Etc.

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Síntesis**
    - ➢ **Optimización del código**

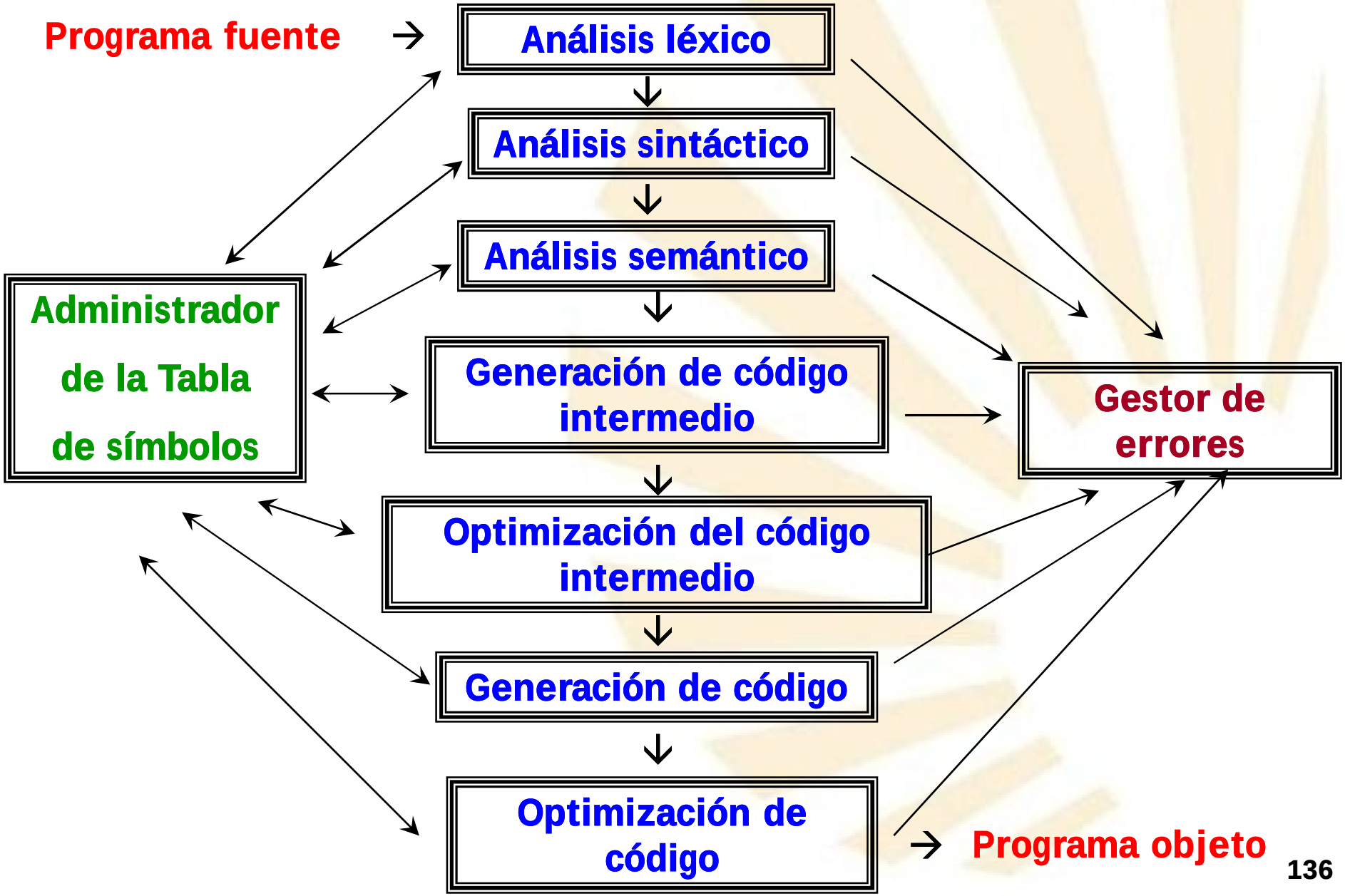
```
100. MOV divisor, R1
101. CMP #0, R1
102. JE 110
103. MOV cociente, R2
104. MUL R2, R1
105. MOV R1, t1
106. MOV t1, R3
107. MOV resto, R4
108. SUM R3, R4
109. MOV R4, dividendo
110.  ...
```



```
100. MOV divisor, R1
101. CMP #0, R1
102. JE 108
103. MOV cociente, R2
104. MUL R2, R1
105.MOV resto, R2
106. SUM R1, R2
107. MOV R2, dividendo
108.  ...
```

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
    - ➢ **Componentes auxiliares**
      - ❑ **Administrador de la tabla de símbolos**
      - ❑ **Gestor de errores**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓**Componentes auxiliares**
    - ➢ **Administrador de la tabla de símbolos**
    - ➢ **Gestor de errores**


Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Administrador de la Tabla de símbolos
Generación de código intermedio
Gestor de errores
Optimización del código intermedio
Generación de código
Optimización de código
Programa objeto

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Componentes auxiliares**
    - ➢ **Administrador de la tabla de símbolos**
      - ❑La tabla de símbolos contiene toda la **información** relacionada con los **identificadores** del programa fuente:
        - Variables y constantes
        - Funciones y procedimientos
        - Parámetros
        - Tipos de datos definidos
        - Etiquetas
        - Etc.

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Componentes auxiliares**
    - ➢ **Administrador de la tabla de símbolos**
      - ❑ **Se crea durante el análisis léxico**
      - ❑ Es completada y utilizada durante **todas las fases** del proceso de compilación
      - ❑ Se puede utilizar **más de una tabla** de símbolos para controlar las reglas de ámbito del lenguaje de programación.
      - ❑ Los **depuradores** pueden mostrar los **valores** de las variables al consultar **la tabla de símbolos**.

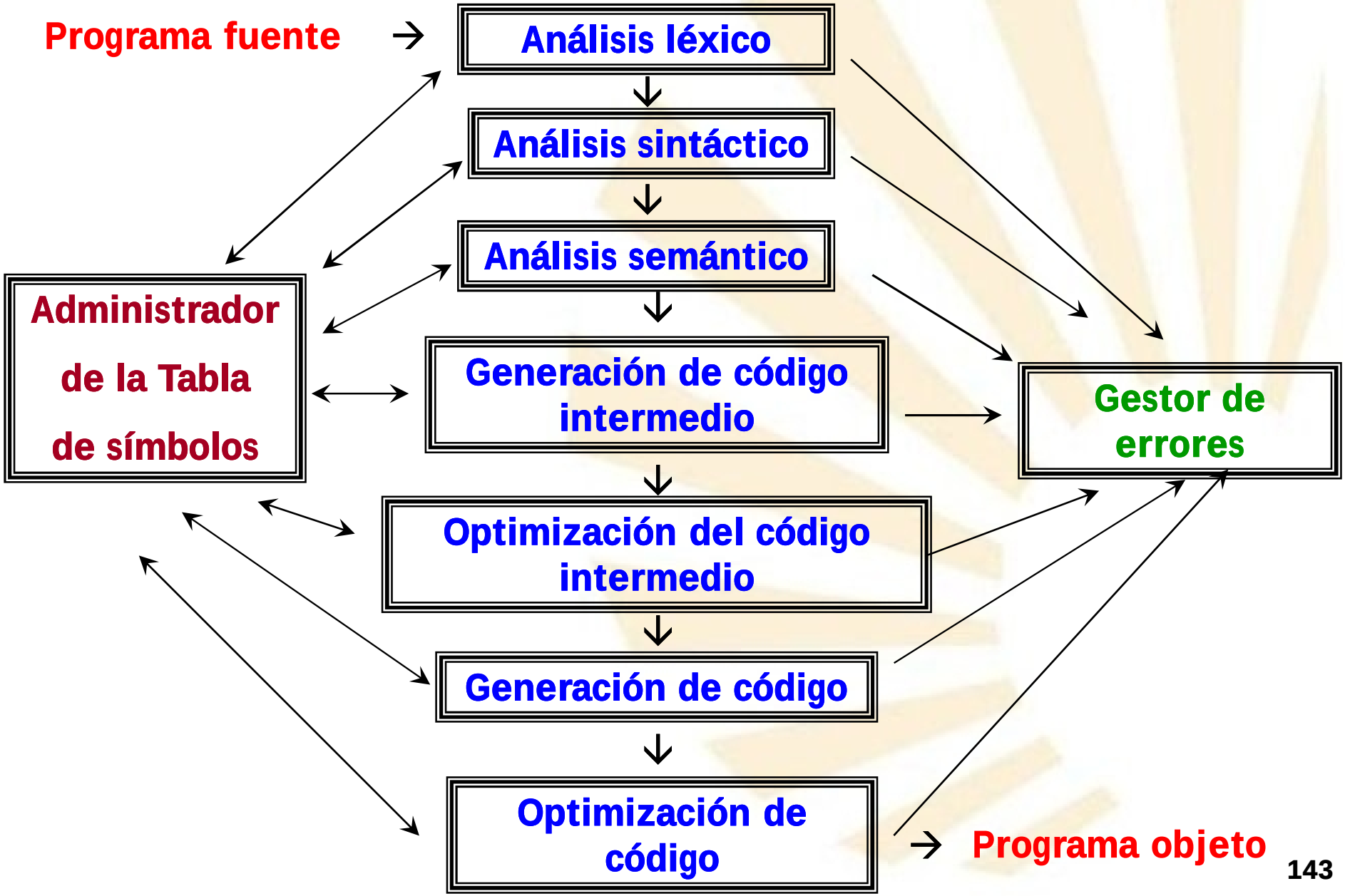
- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Componentes auxiliares**
    - ➢ **Administrador de la tabla de símbolos**
      - ❑V.g.: **dato = 3;**

Nombre	Tipo	Valor	...
dato	entero	3	...

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Componentes auxiliares**
    - ➢ **Administrador de la tabla de símbolos**
      - ❑ La información de las **funciones** o **procedimientos** es más **completa**:
        - Número parámetros
        - Tipo y forma de paso de cada parámetro
        - Tipo de resultados (en las funciones)
        - Etc.

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Componentes auxiliares**
    - ➢ **Administrador de la tabla de símbolos**
      - ❑Las **operaciones** sobre la tabla de símbolos son:
        - Inserción
        - Consulta
        - Modificación
      - ❑Se puede mejora la **eficiencia** en el uso de la tabla de símbolos mediante:
        - Una buena **organización** de la tabla (v.g.: árbol binario de búsqueda)
        - La codificación de las **funciones de acceso** en lenguajes de **bajo nivel** (v.g.: ensamblador).

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Componentes auxiliares**
    - ➢ **Administrador de la tabla de símbolos**
    - ➢ **Gestor de errores**


Programa fuente
Análisis léxico
Análisis sintáctico
Análisis semántico
Administrador de la Tabla de símbolos
Generación de código intermedio
Gestor de errores
Optimización del código intermedio
Generación de código
Optimización de código
Programa objeto

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**

✓ **Componentes auxiliares**

➢ **Gestor de errores**

❑La gestión de errores es un proceso **fundamental**

❑ Hay **errores** en **todas las fases** del proceso de traducción

❑ Errores **más frecuentes** en las etapas de **análisis:**

- **Errores léxicos:** identificador con un carácter no permitido
- **Errores sintácticos:** sentencia de control mal escrita
- **Errores semánticos:** uso de una variable en un contexto inadecuado

❑La gestión de errores **debe**:

- **Informar** sobre el error,
- y permitir, si es posible, que **continúe la traducción** para detectar más errores (**recuperación del error).**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Componentes auxiliares**
    - ➢ **Gestor de errores**
      - ❑ Las **características** de un buen gestor de errores:
        1. Capacidad de **detección** de errores.
        2. **Tratamiento** de los errores.
        3. **Recuperación** del error.
        4. **Evitar la cascada de errores**.
        5. **Información de los errores.**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Componentes auxiliares**
    - ➢ **Gestor de errores**
      - ❑ Las **características** de un buen gestor de errores:
        1. Capacidad de **detección** de errores.
           ❖ Debe ser capaz de reconocer errores de todo tipo: léxico, sintáctico, semántico, etc.
        2. **Tratamiento** de los errores.
        3. **Recuperación** del error.
        4. **Evitar la cascada de errores**.
        5. **Información de los errores**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**

✓ **Componentes auxiliares**

➢ **Gestor de errores**

❑ Las **características** de un buen gestor de errores:

1. Capacidad de **detección** de errores
2. **Tratamiento** de los errores:
   - ❖ Al encontrar un error, intentará subsanarlo si es posible.
   - ❖ Siempre **informará** de los cambios realizados, para que la **persona** que programe **tome la decisión final**.
3. **Recuperación** del error.
4. **Evitar la cascada de errores**.
5. **Información de los errores**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Componentes auxiliares**
    - ➢ **Gestor de errores**
      - ❑ Las **características** de un buen gestor de errores:
        1. Capacidad de **detección** de errores
        2. **Tratamiento** de los errores.
        3. **Recuperación** del error:
           - ❖ Debe permitir que la **traducción continúe**, sobre todo si no se desarrolla en un proceso interactivo.
        4. **Evitar la cascada de errores**.
        5. **Información de los errores.**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Componentes auxiliares**
    - ➢ **Gestor de errores**
      - ❑ Las **características** de un buen gestor de errores:
        1. Capacidad de **detección** de errores
        2. **Tratamiento** de los errores.
        3. **Recuperación** del error.
        4. **Evitar la cascada de errores**
           ❖ Debe **informar** de un error solamente **una vez**, aunque aparezca varias veces, y no generar otros errores.
        5. **Información de los errores.**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Componentes auxiliares**
    - ➢ **Gestor de errores**
      - ❑ Las **características** de un buen gestor de errores :

5. **Información de los errores**: el mensaje de error debe tener las siguientes características:

❖ **Localización**: se debe indicar la línea del código fuente en la que aparece el error.

❖ **Pertinencia**: debe referirse al código del programa y no a detalles internos de la traducción

❖ **Comprensión**: debe ser claro y sencillo

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Fases**
  - ✓ **Pasos**

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Pasos**
    - ➢ Número de **veces** que se **procesa** una representación del **programa fuente**.
    - ➢ Cada paso requiere:
      - ❑ Lectura del código fuente
      - ❑ Procesamiento
      - ❑ Almacenamiento de la información generada
    - ➢ El **número** de **pasos** debe ser **mínimo**.

- **ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS**
  - ✓ **Pasos**
    - ➢ **Las pasadas se suelen agrupar**. Ejemplo:
      - ❑ Primera pasada: análisis léxico, sintáctico, semántico y generación y optimización de código intermedio
      - ❑ Segunda pasada: generación y optimización de código.
    - ➢ **Algunas veces es imprescindible** realizar dos o **más pasos**:
      - ❑ **Algol 68 y PL/I** permiten utilizar las variables antes de ser declaradas.
      - ❑ Si el lenguaje permite **saltos incondicionales** (v.g.: instrucción "goto")
        - ▪ La técnica de "**backpatching**" o "**relleno de retroceso**" permite combinar dos pasadas en una sola. Se requiere una tabla de "saltos".

TEMA I.- INTRODUCCIÓN

- TRADUCCIÓN E INTEPRETACIÓN
- TIPOS DE TRADUCTORES
- PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN
- ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS
- HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES
- COMBINACIÓN DE COMPILADORES: “BOOTSTRAPPING”

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

✓ **Se pueden generar automáticamente algunas partes del proceso de traducción**

✓ **Tipos de herramientas de generación automática:**

- Generadores automáticos de analizadores léxicos
- Generadores automáticos de analizadores sintácticos
- Generadores automáticos de código intermedio
- Generadores automáticos de código
- Máquinas de optimización de código

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

- ✓ **Tipos de herramientas de generación automática:**
  - ➢ **Generadores automáticos de analizadores léxicos**
  - ➢ **Generadores automáticos de analizadores sintácticos**
  - ➢ **Generadores automáticos de código intermedio**
  - ➢ **Generadores automáticos de código**
  - ➢ **Máquinas de optimización de código**

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

✓ **Tipos de herramientas de generación automática:**

➢ **Generadores automáticos de analizadores léxicos**

❑ **Las expresiones regulares** pueden denotar a los componentes básicos de los lenguajes de programación:

- Identificadores
- Números
- Operadores aritméticos, lógicos y relacionales
- Símbolos de puntuación
- Comentarios
- Etc.

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

✓ **Tipos de herramientas de generación automática:**

➢ **Generadores automáticos de analizadores léxicos**

❑ **Las expresiones regulares** pueden denotar a los componentes básicos de los lenguajes de programación.

❑ Existen herramientas automáticas para generar **analizadores léxicos** a partir de las expresiones regulares:

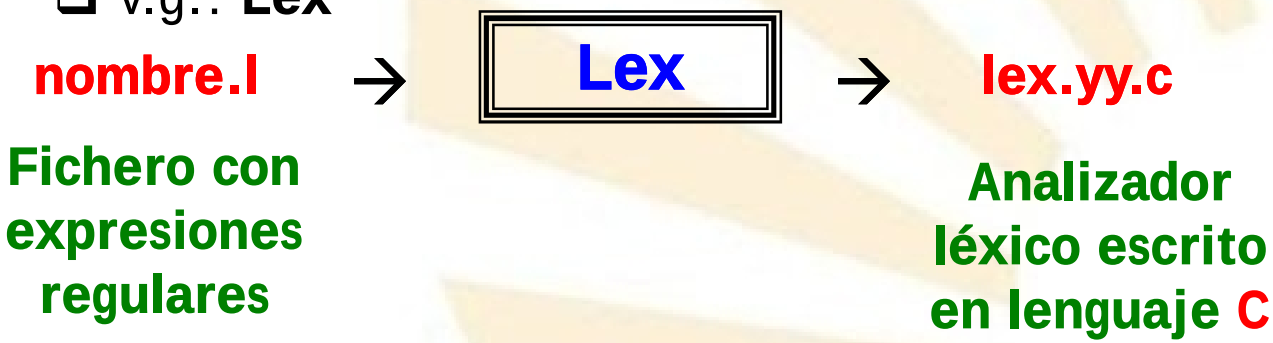
- Lex, Flex, PCLex
- ANTLR
- Etc.

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

✓ **Tipos de herramientas de generación automática:**

➢ **Generadores automáticos de analizadores léxicos**

❑ V.g.: **Lex**



❑ **lex.yy.c:** contiene una función denominada **"yylex()"** que realiza las funciones del **analizador léxico**.

❑ **yylex**(): simula el funcionamiento de un **autómata finito determinista (AFD).**

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

- ✓ **Tipos de herramientas de generación automática:**
  - ➢ **Generadores automáticos de analizadores léxicos**
  - ➢ **Generadores automáticos de analizadores sintácticos**
  - ➢ **Generadores automáticos de código intermedio**
  - ➢ **Generadores automáticos de código**
  - ➢ **Máquinas de optimización de código**

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

- ✓ **Tipos de herramientas de generación automática:**
  - ➢ **Generadores automáticos de analizadores sintácticos**
    - ❑ Las **gramáticas de contexto libre** permiten generar **"casi"** todas las **estructuras sintácticas** de los lenguajes de programación.
    - ❑ Generadores:
      - YACC o Bison
      - LLGEN
      - CUP
      - ANTLR
      - Etc.

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

✓ **Tipos de herramientas de generación automática:**

➢ **Generadores automáticos de analizadores sintácticos**

❑V.g.: **YACC**, **Y**et **A**nother **C**ompiler **C**ompiler

❑ **y.tab.c:** contiene una función denominada **"yyparse()"** que realiza las funciones de analizador sintático.

❑ **yyparse():** simula el funcionamiento de **un autómata con pila.**

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

- ✓ **Tipos de herramientas de generación automática:**
  - ➢ **Generadores automáticos de analizadores léxicos**
  - ➢ **Generadores automáticos de analizadores sintácticos**
  - ➢ **Generadores automáticos de código intermedio**
  - ➢ **Generadores automáticos de código**
  - ➢ **Máquinas de optimización de código**

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

- ✓ **Tipos de herramientas de generación automática:**
  - ➢ **Generadores automáticos de código intermedio**
    - ❑ **Suelen estar integrados en los analizadores sintácticos**
    - ❑ **Hay dos versiones** que incorporan **acciones semánticas de generación de código intermedio**:
      - ▪ **Definiciones basadas en la sintaxis**
      - ▪ **Esquemas de traducción**
    - ❑ **Al crear el árbol sintáctico**, se **ejecutan** las **acciones semánticas** de generación de código intermedio

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

- ✓ **Tipos de herramientas de generación automática:**
  - ➢ **Generadores automáticos de analizadores léxicos**
  - ➢ **Generadores automáticos de analizadores sintácticos**
  - ➢ **Generadores automáticos de código intermedio**
  - ➢ **Generadores automáticos de código**
  - ➢ **Máquinas de optimización de código**

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

- ✓ **Tipos de herramientas de generación automática:**
  - ➢ **Generadores automáticos de código**
    - ❑ Se utilizan **transformaciones basadas en reglas** que tienen en cuenta:
      - ▪ Las **características** de las **sentencias** y **operaciones** del código intermedio
      - ▪ Las características de la máquina donde se va a ejecutar el código objeto: acceso a datos, operaciones básicas
      - ▪ Las reglas utilizan **plantillas** de conversión.

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

- ✓ **Tipos de herramientas de generación automática:**
  - ➢ **Generadores automáticos de analizadores léxicos**
  - ➢ **Generadores automáticos de analizadores sintácticos**
  - ➢ **Generadores automáticos de código intermedio**
  - ➢ **Generadores automáticos de código**
  - ➢ **Máquinas de optimización de código**

• HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES

- ✓ **Tipos de herramientas de generación automática:**
  - ➢ **Máquinas de optimización de código**
    - ❑ Se utilizan **dispositivos para el análisis del flujo de datos**.
    - ❑ Se recoge **información** sobre la forma en que los **valores** se **transmiten** de una parte a otra del programa
    - ❑ Ejemplos:
      - ▪ Análisis de "**uso siguiente**" o de "vida": se comprueba en qué lugares se usa una variable y, especialmente, cuándo no se va a utilizar más.
      - ▪ Si una **variable** es utilizada **frecuentemente** entonces es preferible almacenarla en un **registro de máquina**.
      - ▪ Etc.

TEMA I.- INTRODUCCIÓN

- TRADUCCIÓN E INTEPRETACIÓN
- TIPOS DE TRADUCTORES
- PROGRAMAS RELACIONADOS CON LA TRADUCCIÓN
- ESTRUCTURA DE UN COMPILADOR: FASES Y PASOS
- HERRAMIENTAS PARA LA CONSTRUCCIÓN DE COMPILADORES
- COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"

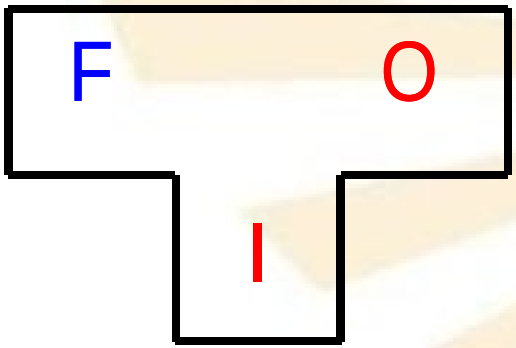
- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**

  ✓ La técnica de "**bootstrapping**" permite **combinar compiladores** ya **creados** para **construir nuevos compiladores**

  ✓ Los lenguajes que aparecen en el proceso de compilación son:

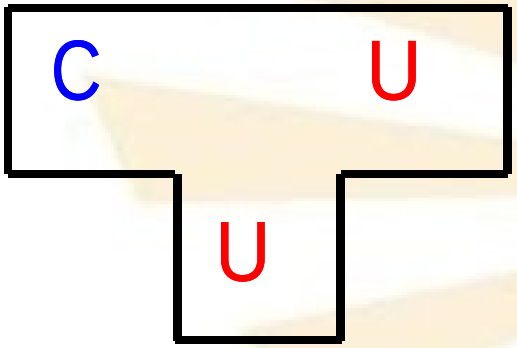
  - Lenguaje **fuente** (F)
  - Lenguaje de **implementación** (I): lenguaje en el que está escrito el compilador
  - Lenguaje **objeto** (O)

  ✓ El compilador se puede representar en forma de T

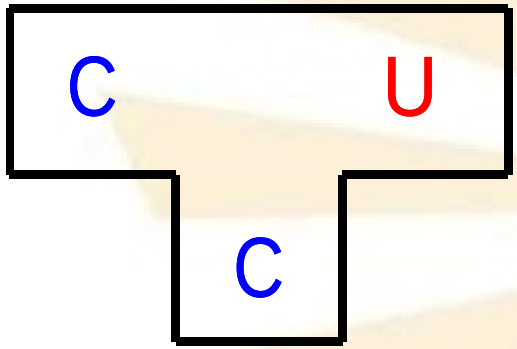


✓**Nota**: si el lenguaje es ejecutable, se indicará con color rojo

- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ **Ejemplo**: compilador **gcc**
    - ➢ Lenguaje fuente: lenguaje C
    - ➢ Lenguaje de implementación: lenguaje máquina de Unix (U)
    - ➢ Lenguaje objeto: lenguaje máquina de Unix (U)



- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Si **F = I** entonces el compilador se denomina **"autocompilador"**
  - ✓ **Ejemplo**:
    - ➢ Lenguaje fuente: Lenguaje C
    - ➢ Lenguaje de implementación: Lenguaje máquina de Unix (U)
    - ➢ Lenguaje objeto: Lenguaje C



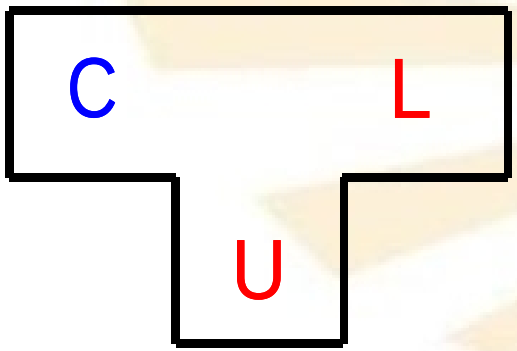
➢ **Nota**: habría que "compilar" este compilador para que se pueda ejecutar, porque el lenguaje de implementación **no** es ejecutable.

- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**

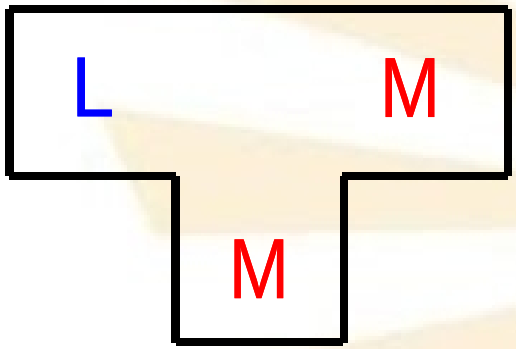
  ✓ Si **I ≠ O** entonces el compilador se denomina **"compilador cruzado"**, porque se genera código para una máquina diferente a la que se ha compilado

  ✓ Ejemplo:

  - Lenguaje fuente: Lenguaje C
  - Lenguaje de implementación: Lenguaje máquina de Unix (U)
  - Lenguaje objeto: Lenguaje máquina de Linux (L)



- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Ejemplo 1**: se pretende construir el siguiente compilador
      - ❑ Lenguaje fuente: Lenguaje L de alto nivel
      - ❑ Lenguaje de implementación: Lenguaje máquina (M)
      - ❑ Lenguaje objeto: Lenguaje máquina (M)



    - ➢ **Dificultad**: es muy difícil escribir un programa (compilador) directamente en código máquina

- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Ejemplo 1**
      - ❑ **Paso 1:** se construyen dos compiladores auxiliares

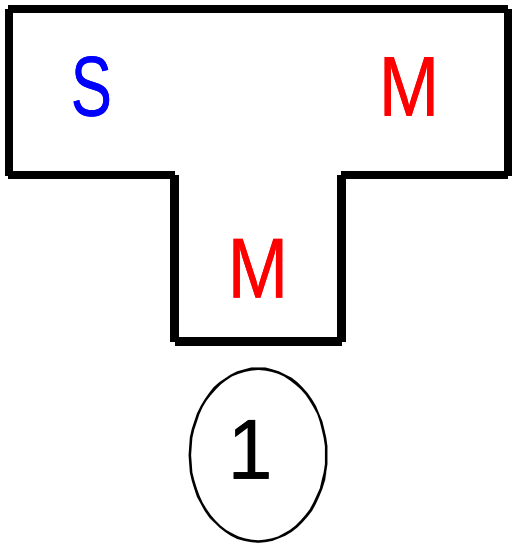
- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**

✓ Aplicación de la técnica de **"bootstrapping"**:

➢ **Ejemplo 1**

❑**Paso 1**: se construyen dos compiladores auxiliares

▪ **Primer compilador**



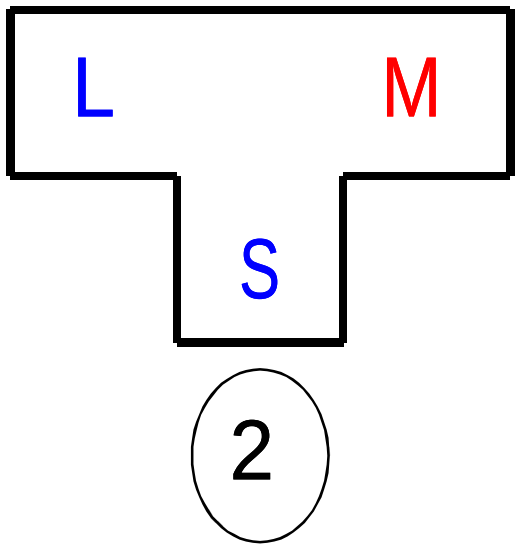
Lenguaje fuente: Lenguaje **S**, que es más simple que el lenguaje L de alto nivel (V.g: un subconjunto de L o ensamblador).

Lenguaje de implementación: Lenguaje máquina (M)

Lenguaje objeto: Lenguaje máquina (M)

Observación: este compilador se puede construir con más **facilidad** porque S es más simple que L.

- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de "**bootstrapping**":
    - ➢ **Ejemplo 1**
      - ❑ **Paso 1:** se construyen dos compiladores auxiliares
        - ▪**Segundo compilador**

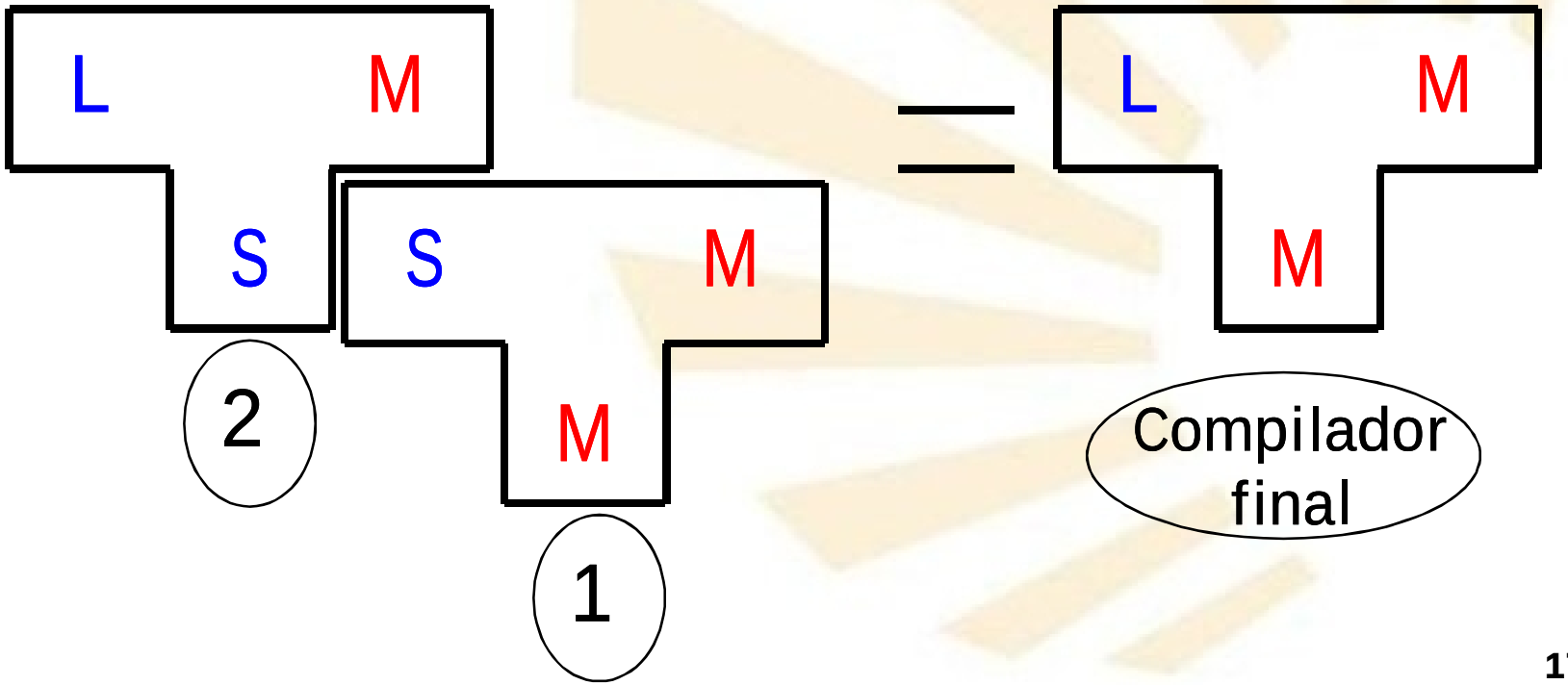


Lenguaje fuente: Lenguaje L de alto nivel

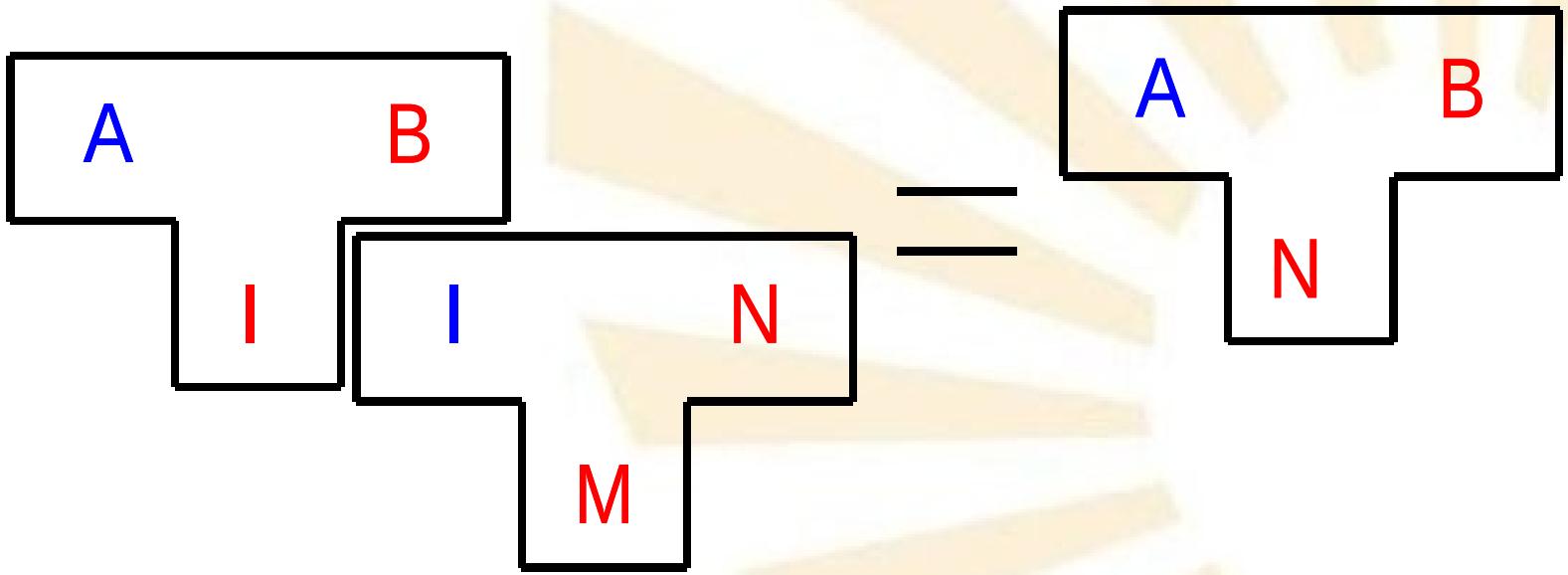
Lenguaje de implementación: Lenguaje S (que es más simple que el lenguaje L de alto nivel)

Lenguaje objeto: lenguaje máquina (M)

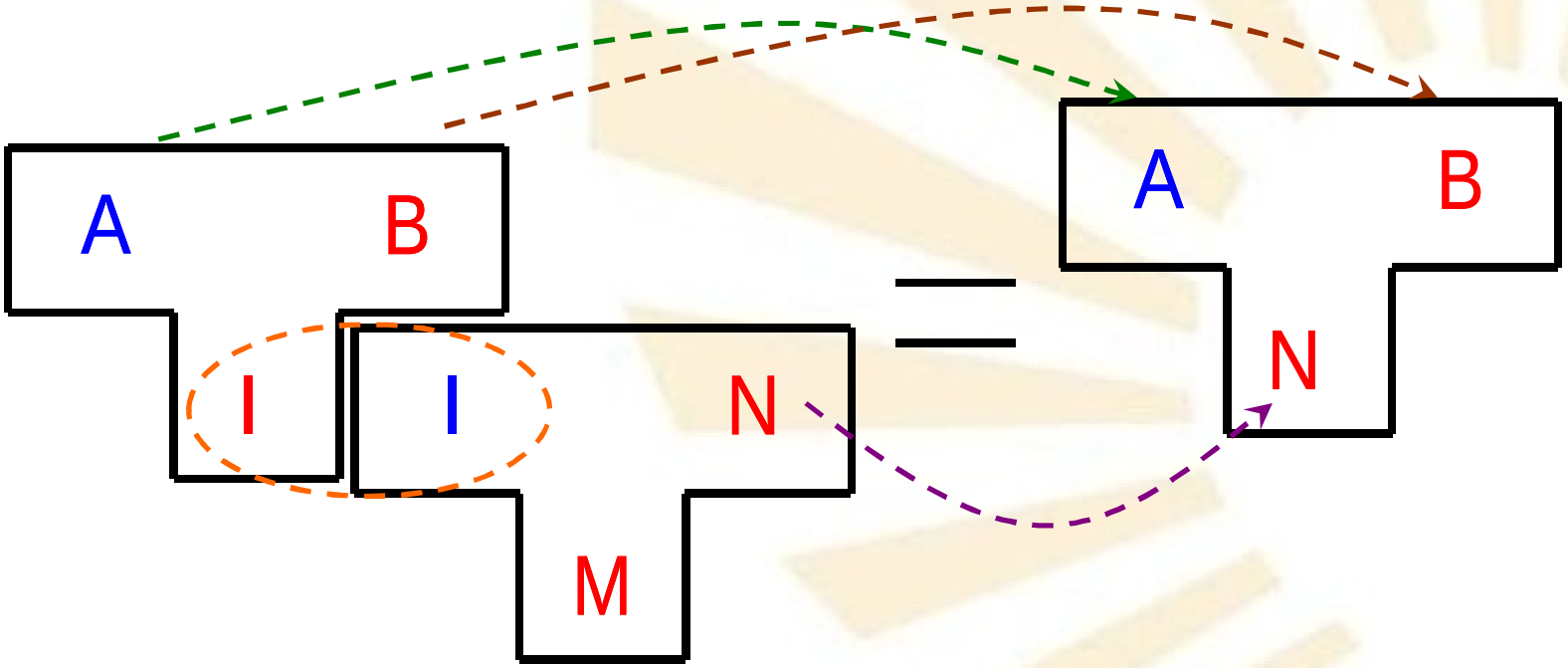
- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de "**bootstrapping**":
    - ➢ **Ejemplo 1**
      - ❑ **Paso 2:** se **compila** el compilador **2** con el compilador **1**, creándose el compilador **final**



- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ Forma general: $A_I B + I_M N = A_N B$



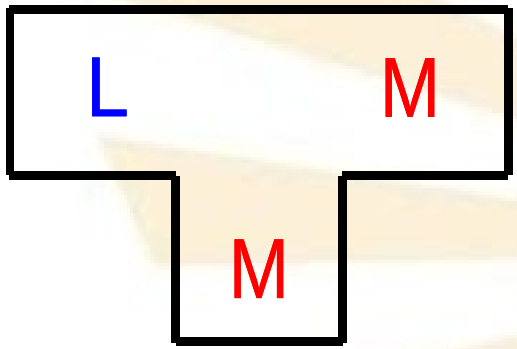
- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ Forma general: $A_I B + I_M N = A_N B$



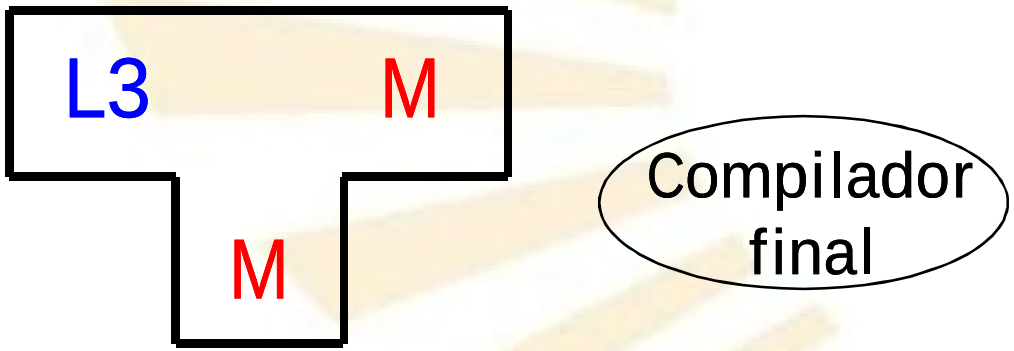
- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**

  ✓ Aplicación de la técnica de **"bootstrapping"**:

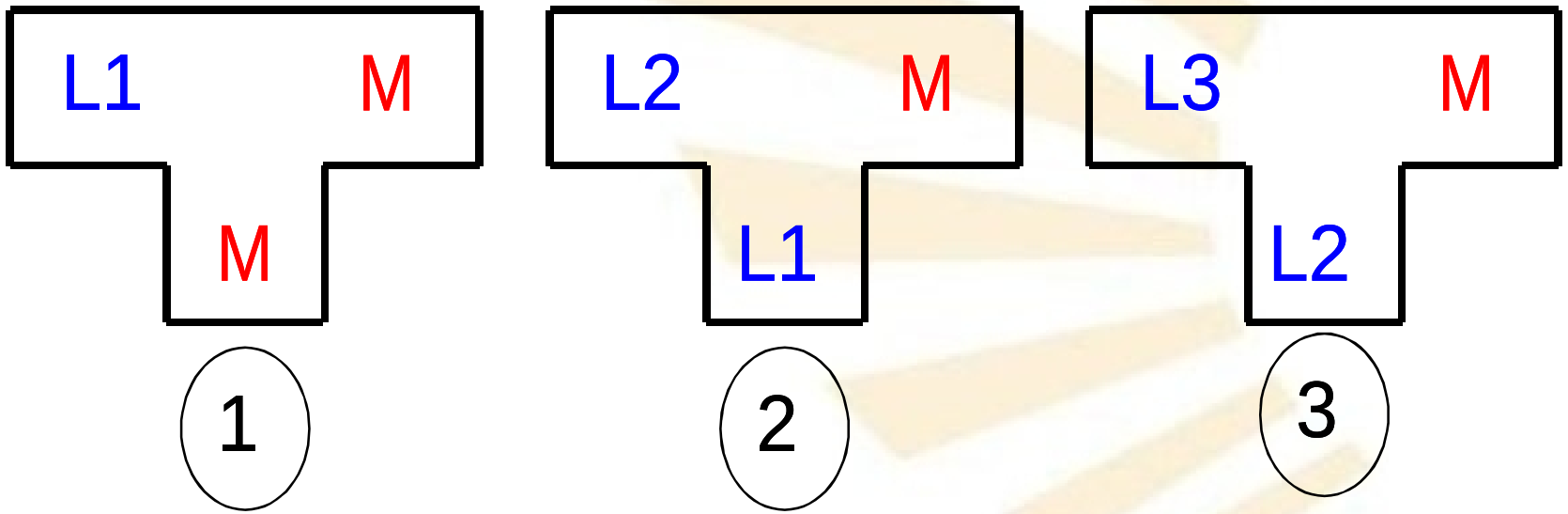
  ➢ Si se desea construir un compilador escrito en un lenguaje máquina **M** para un lenguaje de alto nivel **L**, entonces se utilizan subconjuntos del lenguaje inicial.



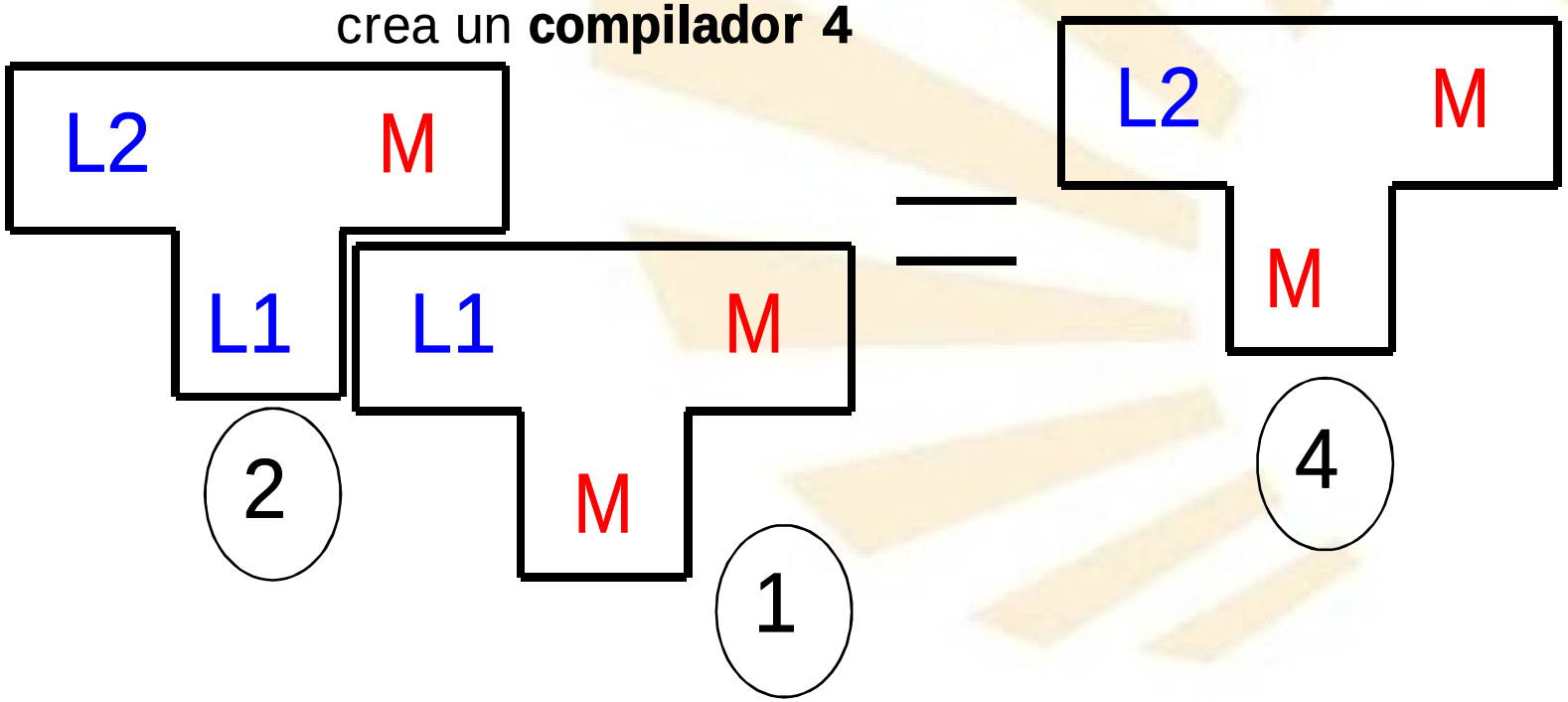
- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Ejemplo 2**
      - ❑ Sean tres lenguajes de programación: **L1 ⊆ L2 ⊆ L3**
      - ❑ Se pretende construir el siguiente compilador
        - Lenguaje fuente: Lenguaje L3 de alto nivel
        - Lenguaje de implementación: Lenguaje máquina (M)
        - Lenguaje objeto: Lenguaje máquina (M)



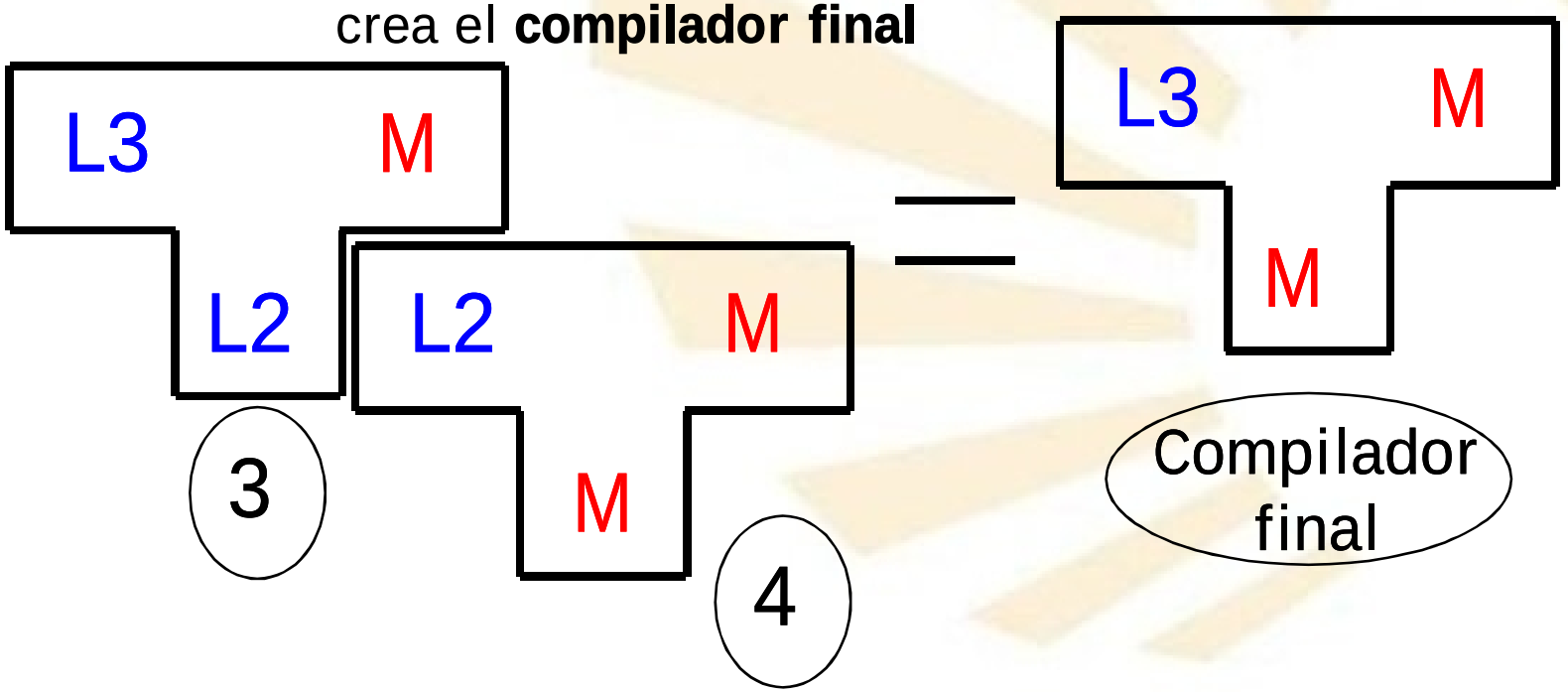
- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Ejemplo 2:**
      - ❑**Paso 1**:
        - ▪Se construyen los siguientes tres compiladores



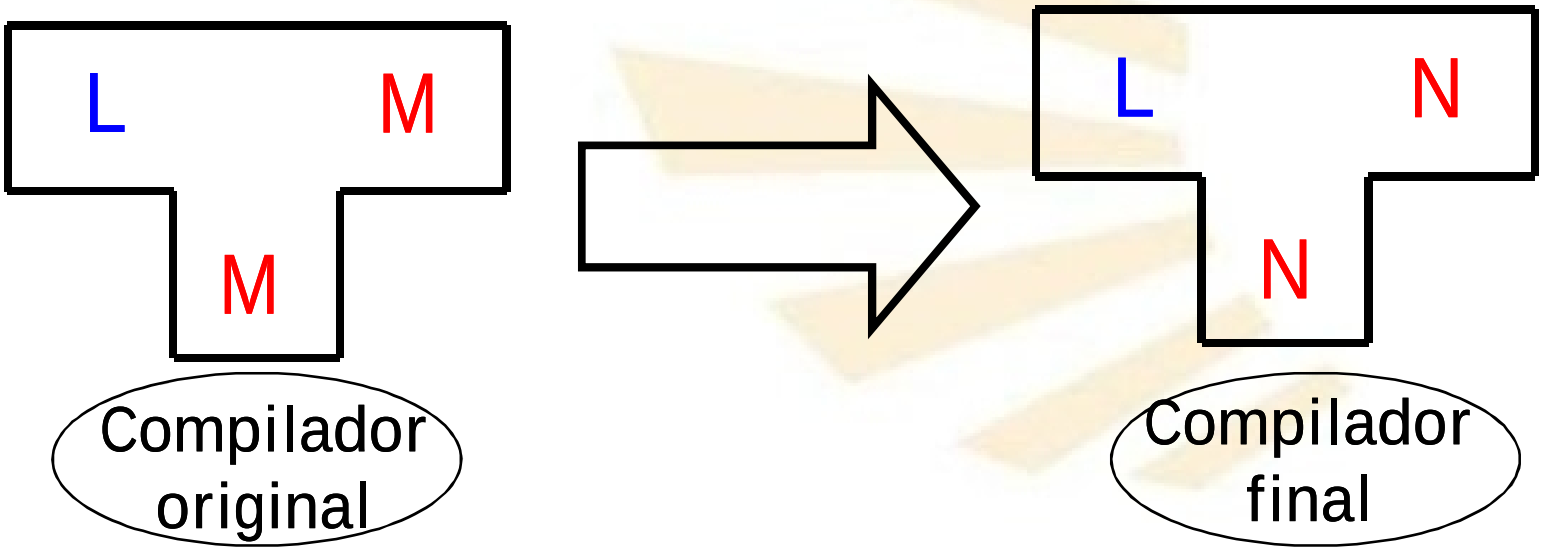
- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Ejemplo 2**
      - ❑ **Paso 2:**
        - ▪ Se compila el **compilador 2** con el **compilador 1**: se crea un **compilador 4**



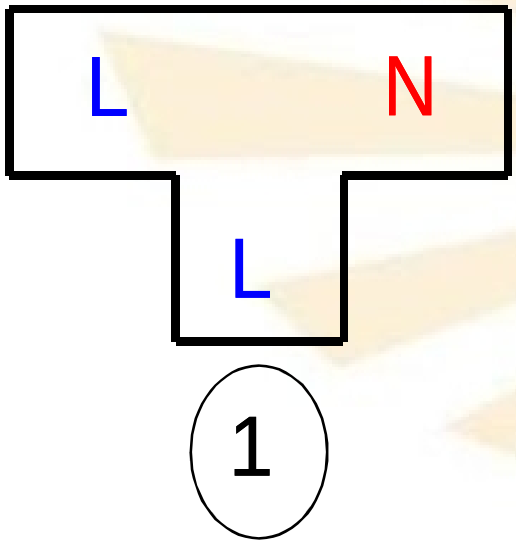
- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Ejemplo2**
      - ❑ **Paso 3:**
        - ▪Se compila el **compilador 3** con el **compilador 4**: se crea el **compilador final**



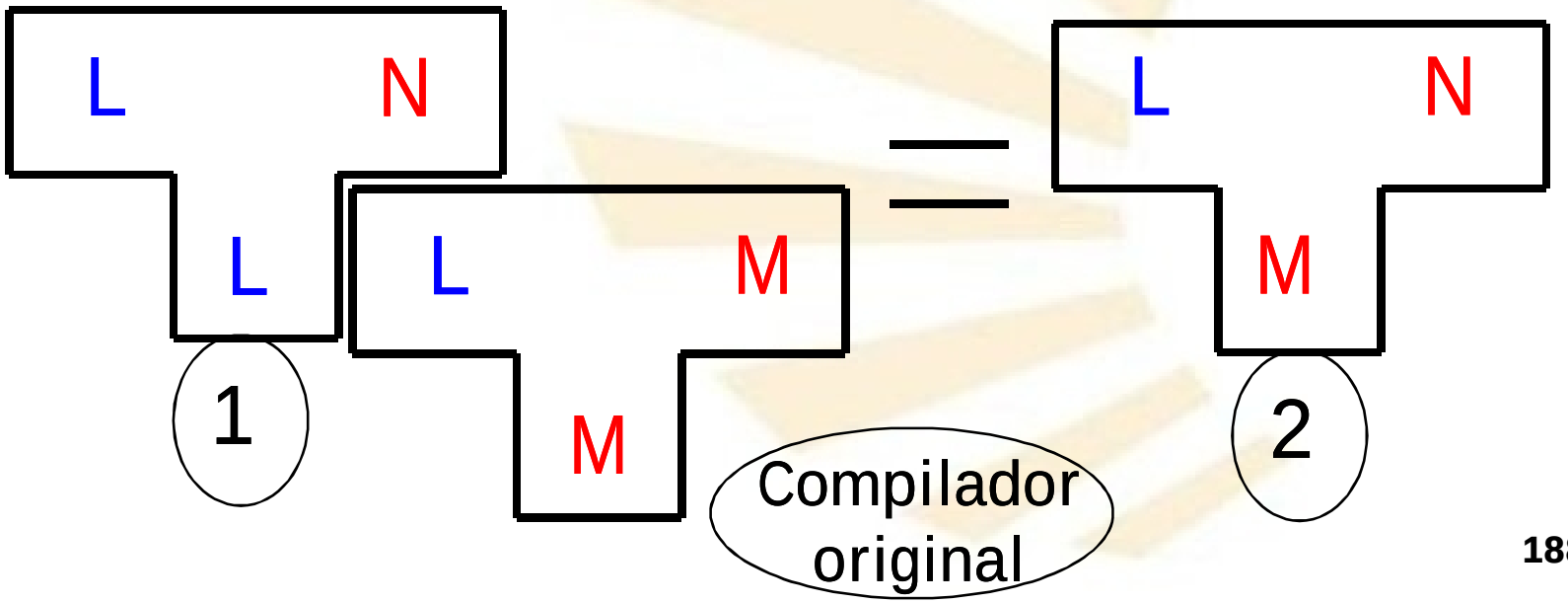
- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Ejemplo 3**:
      - ❑ Dado un compilador de un lenguaje L para una máquina M, se quiere construir otro compilador para otra máquina N
      - ❑ Compilador original: $\mathbf{L}_{\mathbf{M}}\,\mathbf{M}$
      - ❑ Objetivo: $\mathbf{L}_{\mathbf{N}}\,\mathbf{N}$



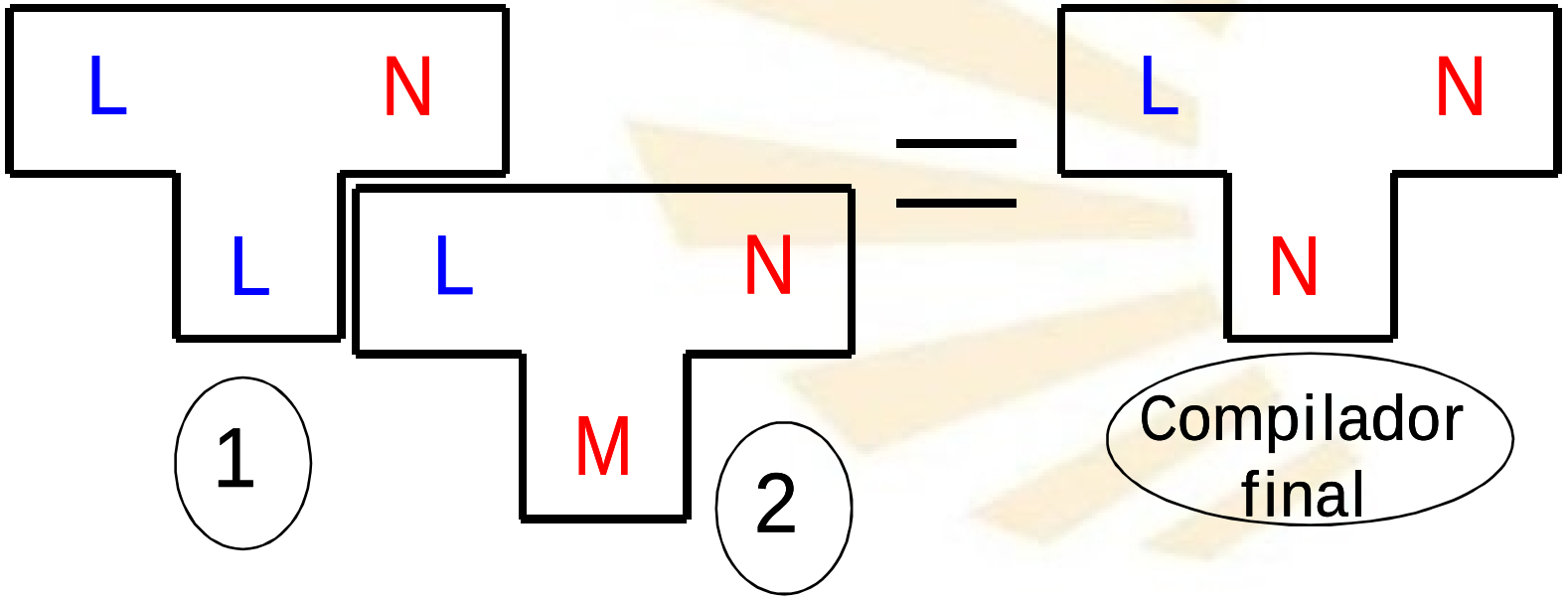
- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Ejemplo 3**
      - ❑ **Paso 1**:
        - ▪Se construye el auto compilador **L_L N**
        - ▪ Este compilador es más fácil de construir que el compilador L_N N



- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Ejemplo 3**
      - ❑ **Paso 2**:
        - ▪Se compila el compilador obtenido en el paso **1** con el compilador **original**: se genera el compilador **cruzado** $L_M N$

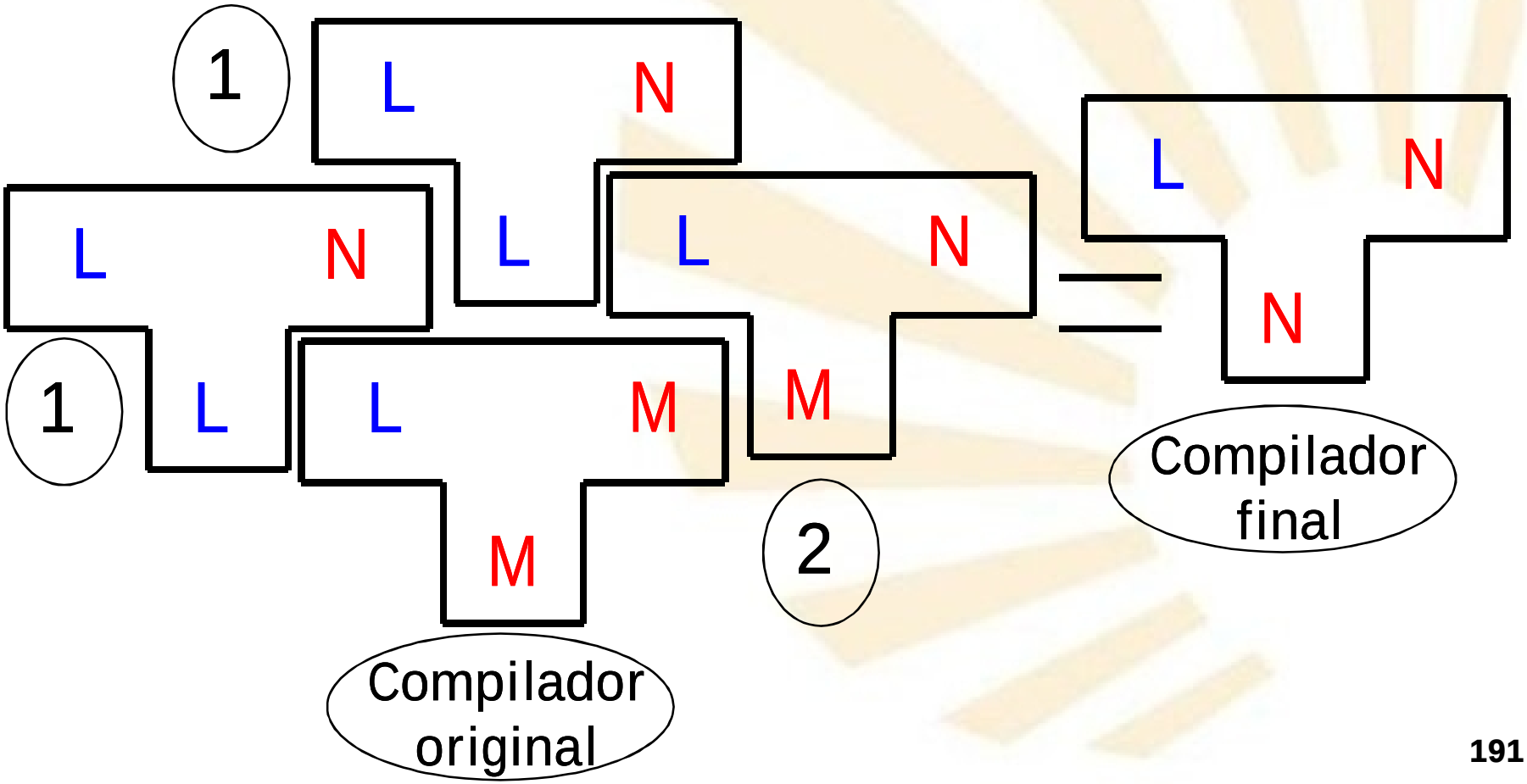


- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Ejemplo 3**
      - ❑ **Paso 3**:
        - ▪Se compila el compilador obtenido en el paso **1** con el compilador obtenido en el paso **2**, creándose el compilador **final**



- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Resumen del ejemplo 3**:
      - ❑ El compilador **original** y el compilador **1** se construyen **directamente**
      - ❑ El compilador **2** se construye a partir del **original** y el compilador **1**
      - ❑ El compilador **final** se construye a partir de los compiladores **1** y **2**

- **COMBINACIÓN DE COMPILADORES: "BOOTSTRAPPING"**
  - ✓ Aplicación de la técnica de **"bootstrapping"**:
    - ➢ **Resumen del ejemplo 3**:



UNIVERSIDAD DE CÓRDOBA

ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE INFORMÁTICA Y ANÁLISIS NUMÉRICO

PROCESADORES DE LENGUAJES

GRADO EN INGENIERÍA INFORMÁTICA

ESPECIALIDAD DE COMPUTACIÓN

TERCER CURSO

SEGUNDO CUATRIMESTRE