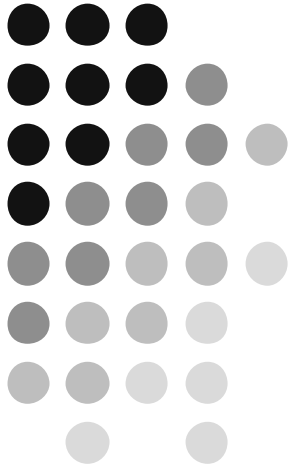
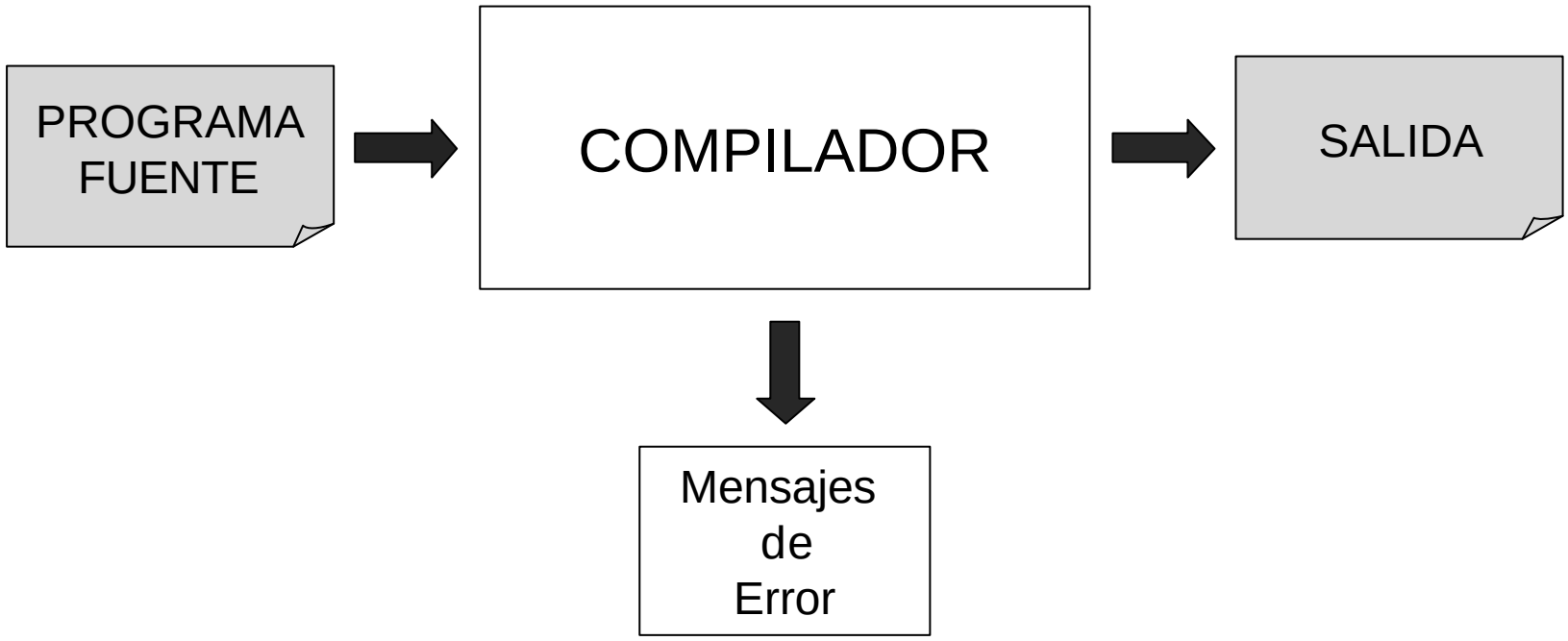
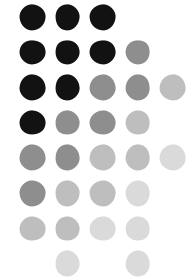
Diseño de Compiladores I

Estructura General de un Compilador



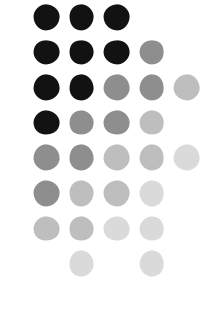
Estructura General de un Compilador





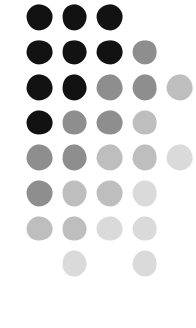
Un compilador es un programa que traduce un programa escrito en lenguaje fuente y produce otro equivalente escrito en un lenguaje destino.

Lenguaje Fuente



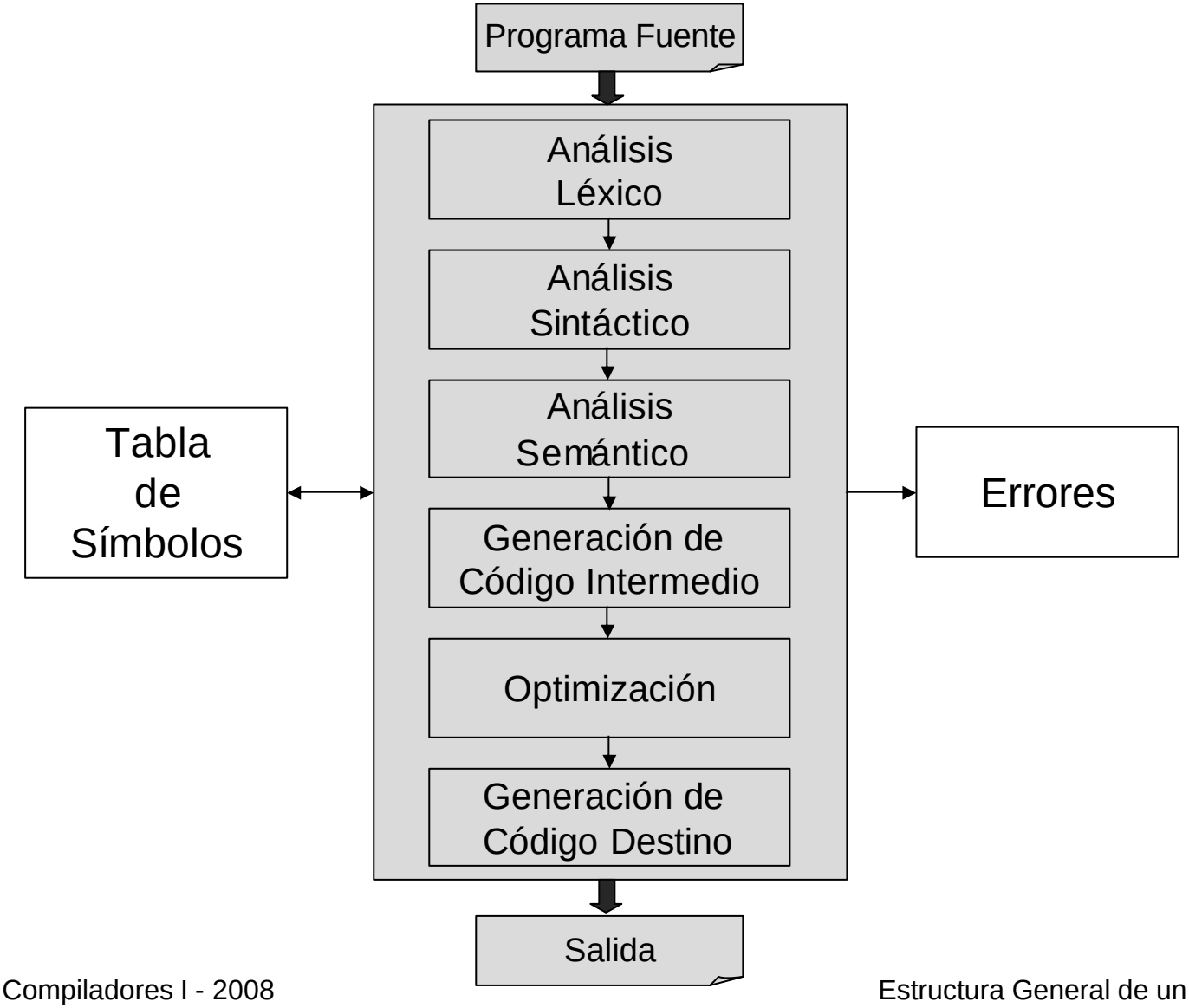
- Lenguaje de alto nivel. Por ejemplo: C, Pascal, C++.
- Lenguaje especializado para alguna disciplina específica dentro de las Ciencias de la Computación.

Salida

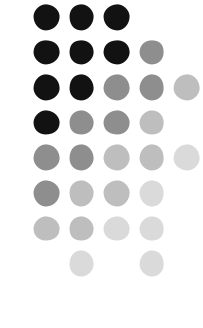


- Código Assembler. Deberá ser ensamblado y vinculado.
- Código Binario. Deberá ser vinculado con las librerías correspondientes para obtener el código ejecutable.
- Código de Máquina. Escrito en las instrucciones de máquina de la computadora en la que se ejecutará.
- Otro lenguaje de alto nivel.

Fases de la Compilación



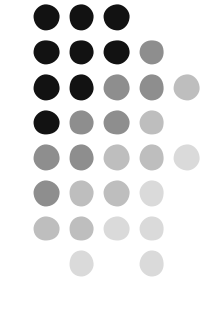
Front End (Análisis)



Fases que dependen del lenguaje fuente

- Análisis Léxico
- Análisis Sintáctico
- Análisis Semántico (Estático)
- Creación de la Tabla de Símbolos
- Generación de Código Intermedio
- Algo de Optimización
- Manejo de errores correspondiente a las fases del Front End

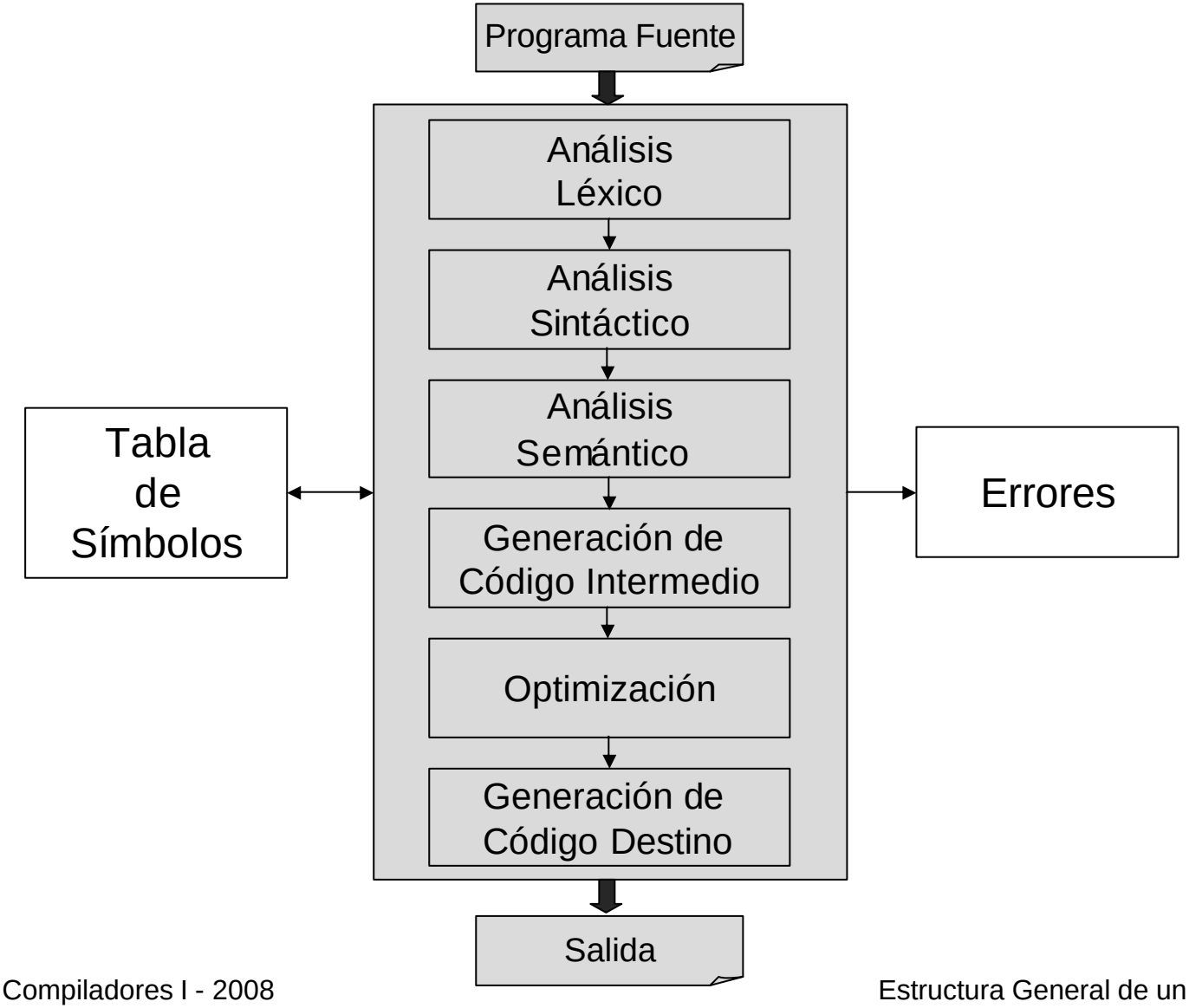
Back End (Síntesis)



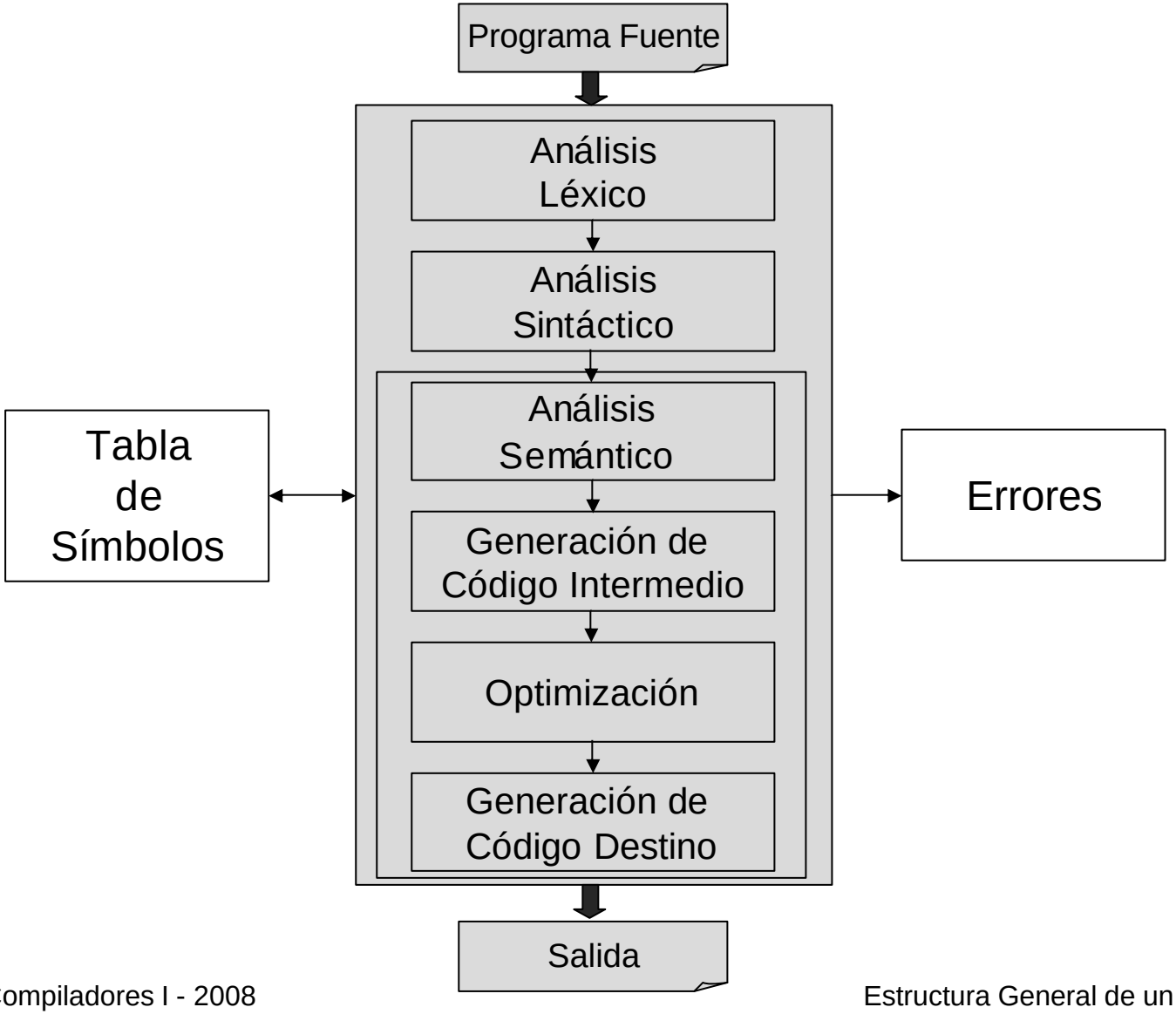
Fases que dependen de la máquina destino

- Generación de la salida
- Optimización
- Manejo de errores correspondiente a las fases del Back End
- Operaciones sobre la Tabla de Símbolos

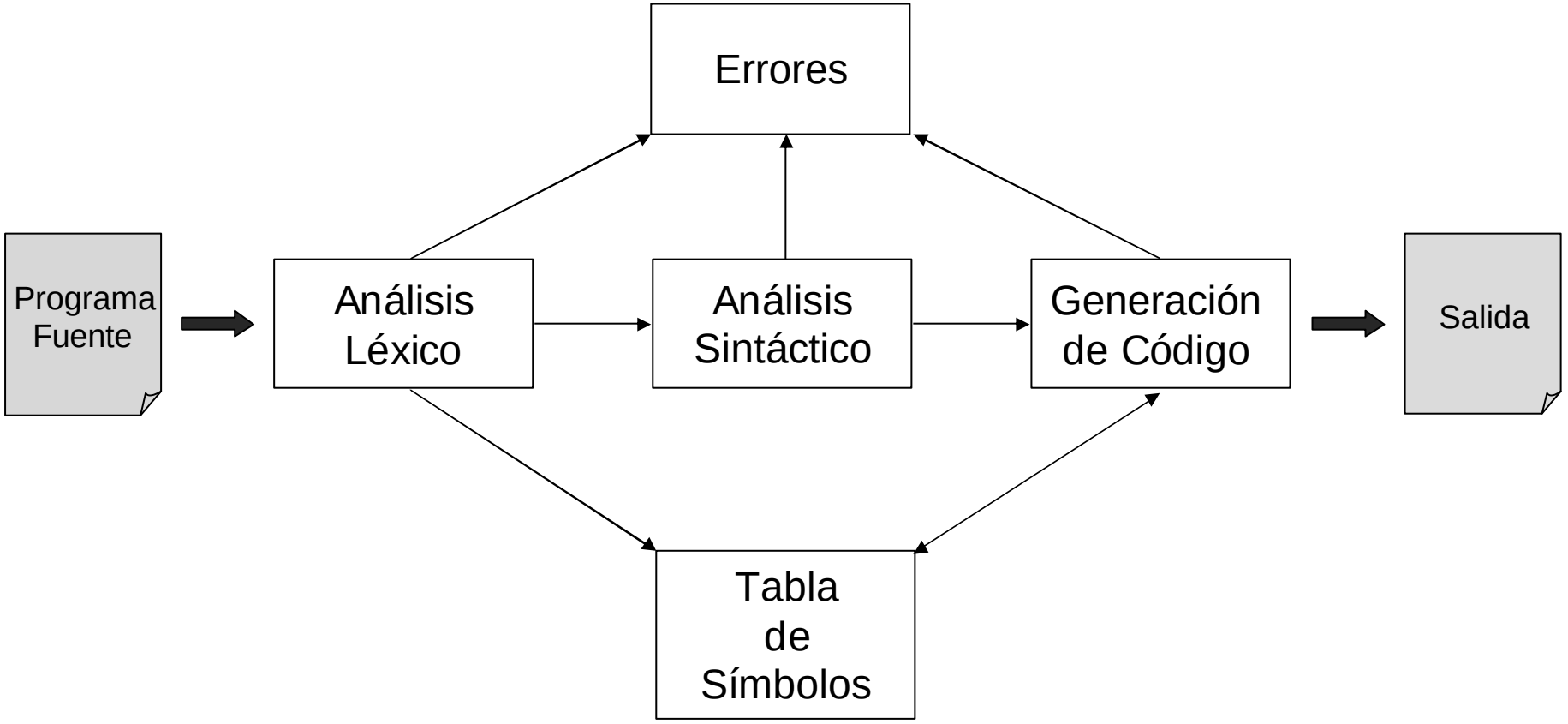
Fases de la Compilación



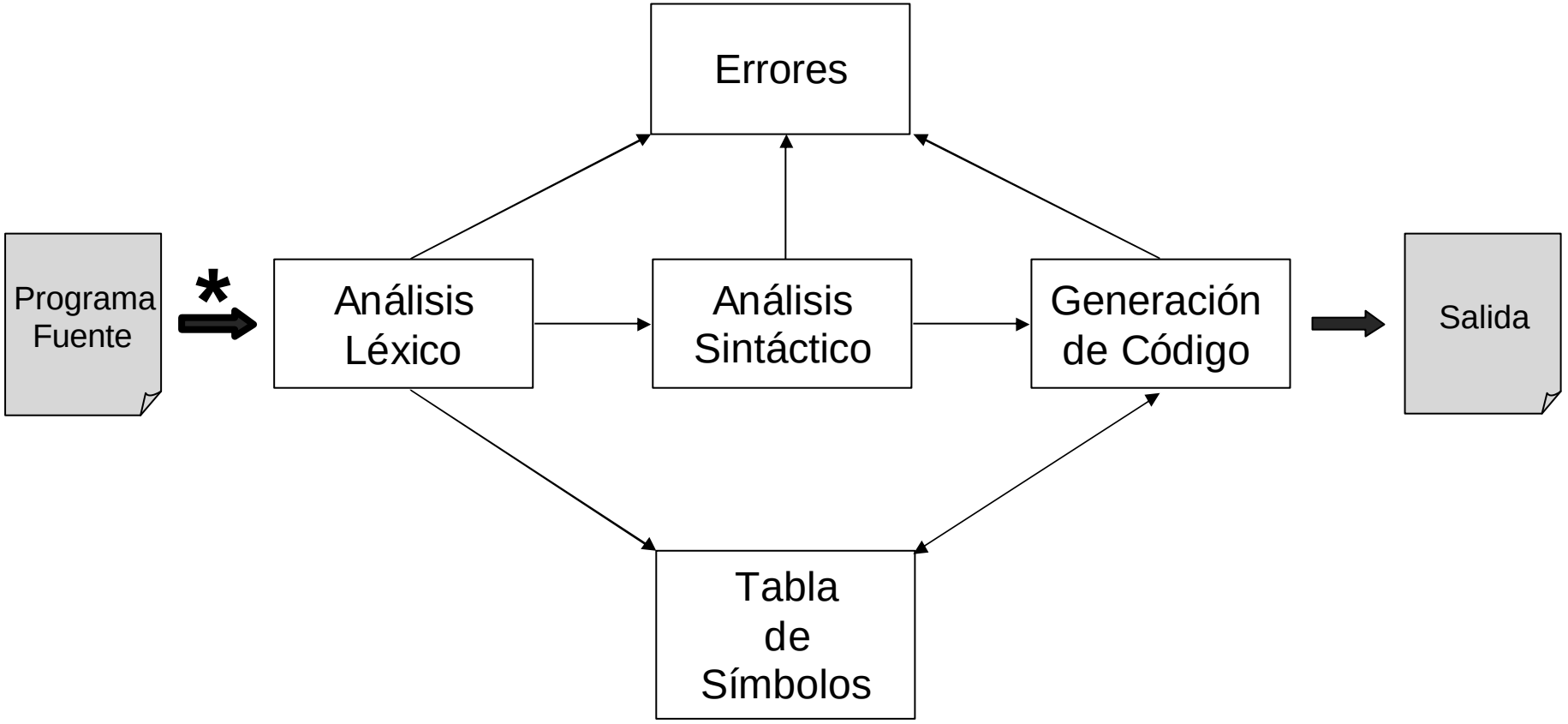
Fases de la Compilación



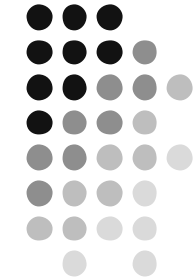
Fases de la Compilación



Fases de la Compilación



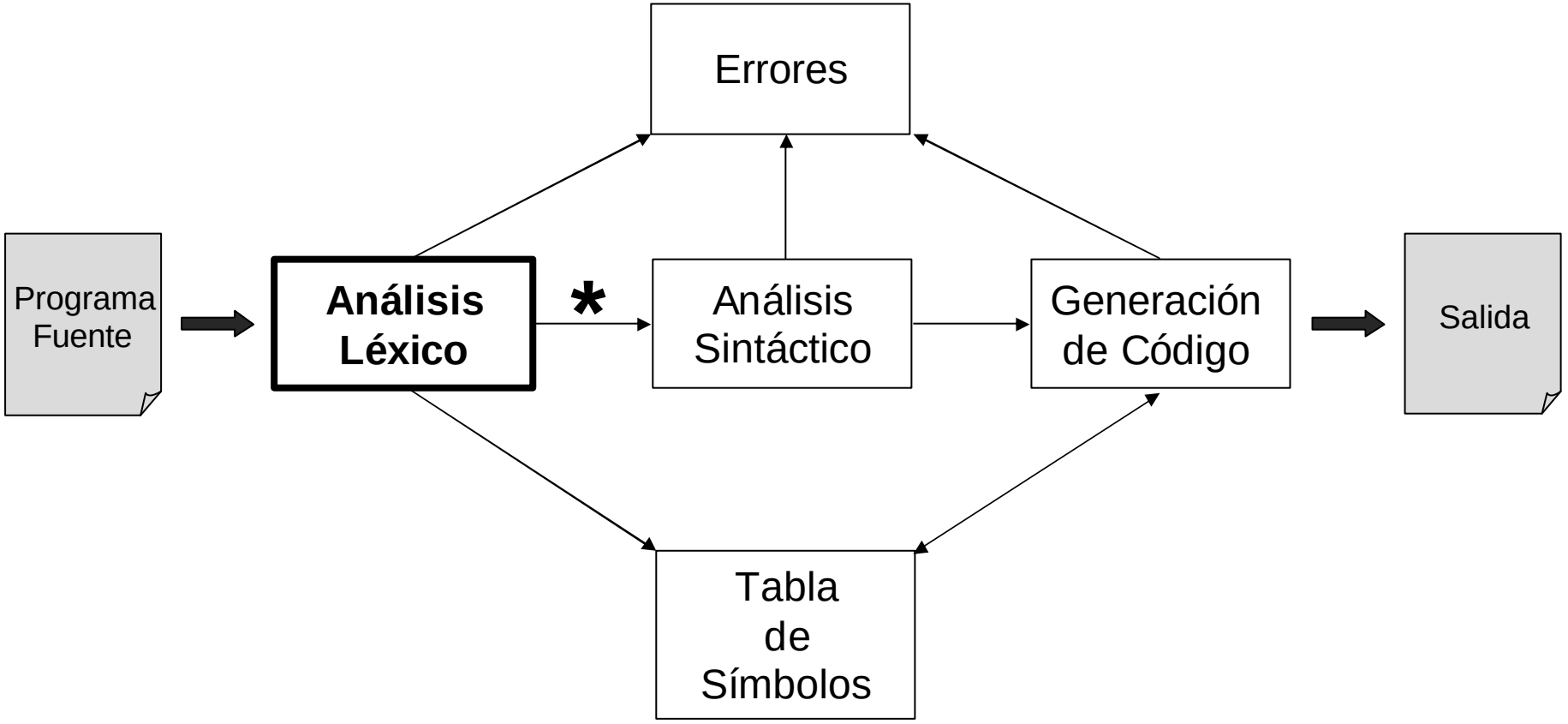
Fases de la Compilación



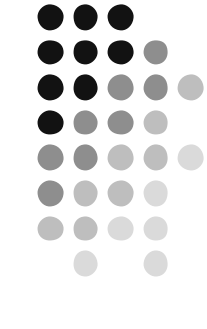
Suele haber preprocesadores para:

- Eliminar comentarios
- Incluir archivos
- Expandir macros
- Efectuar compilación condicional
- Reemplazar constantes simbólicas

Fases de la Compilación

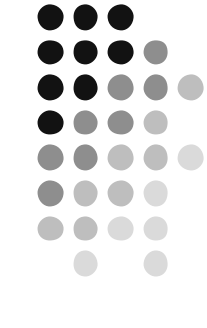


Análisis Léxico



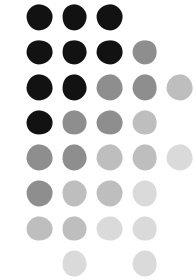
- Lee el programa fuente.
- Remueve espacios en blanco, tabulaciones, saltos de línea.
- Remueve comentarios.
- Agrupa los caracteres en unidades llamadas ***tokens***.

Análisis Léxico



Un ***token*** es una secuencia de caracteres que forman una unidad significativa

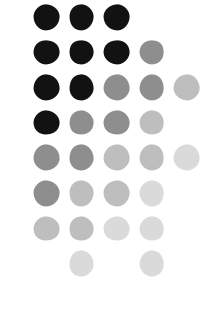
Análisis Léxico



La interacción entre el Análisis Léxico y el Análisis Sintáctico puede ocurrir de distintas formas:

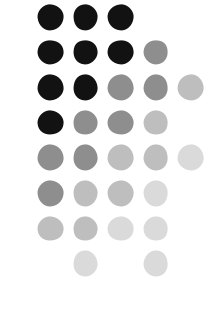
- Ambas actividades se ejecutan en modo batch.
- Ambas actividades son concurrentes.
- Ambas actividades son rutinas del Generador de Código.
- El Análisis Léxico es una rutina del Análisis Sintáctico.

Análisis Léxico
Ejemplo



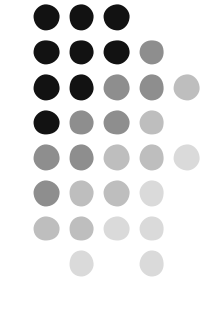
```
if Plazo >= 30
   then Tasa := Base + Recargo / 100
   else Tasa := Base
```

Análisis Léxico
Ejemplo



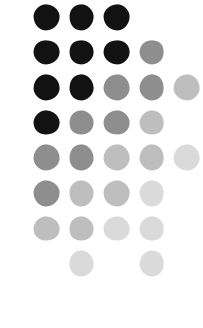
[if] [Plazo] [>=] [30]
    [then] [Tasa] [:=] [Base] [+] [Recargo] [/] [100]
    [else] [Tasa] [:=] [Base]

Análisis Léxico
Tokens



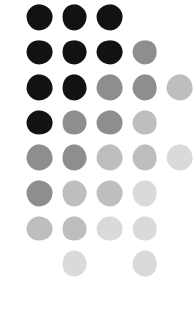
- Palabras reservadas.
  - Ejemplos: IF, THEN, ELSE
- Operadores
  - Ejemplos: ‘+’, ‘>=‘, ‘:=‘
- Cadenas de múltiples caracteres
  - Ejemplos: Identificador, Constante

Análisis Léxico
Tokens



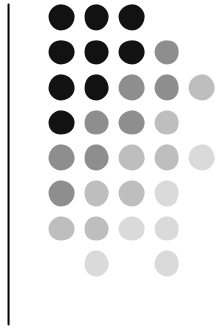
- Los tokens se diferencian de la cadena de caracteres que representan.
- La cadena de caracteres es el ***Lexema*** o valor léxico.
  - Existen tokens que se corresponden con un único lexema
    - Ejemplo: Palabra Reservada IF
  - Existen tokens que pueden representar lexemas diferentes
    - Ejemplo: Identificador Plazo, Identificador Tasa

Análisis Léxico



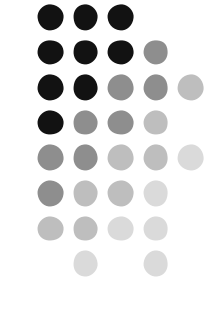
- El Análisis Léxico hace una correspondencia entre cada token y un número entero.
- El Análisis Léxico entrega al Análisis Sintáctico los tokens.
- Cuando un token puede corresponder a más de un lexema, el Análisis Léxico entrega al Análisis Sintáctico el par token-atributo.

Análisis Léxico



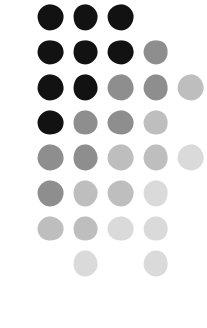
Token	Identificación del token
ID	27
CTE	28
IF	59
THEN	60
ELSE	61
+	70
/	73
>=	80
:=	85

Análisis Léxico
Ejemplo



[if] [Plazo] [>=] [30]
    [then] [Tasa] [:=] [Base] [+] [Recargo] [/] [100]
    [else] [Tasa] [:=] [Base]

Análisis Léxico
Ejemplo

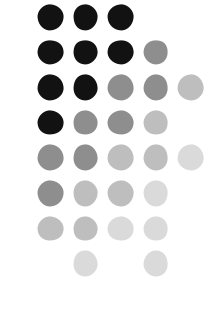


[59] [Plazo] [>=] [30]

[then] [Tasa] [:=] [Base] [+] [Recargo] [/] [100]

[else] [Tasa] [:=] [Base]

Análisis Léxico
Ejemplo

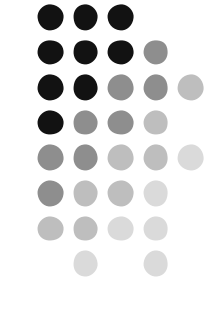


[59] [27] [>=] [30]

[then] [Tasa] [:=] [Base] [+] [Recargo] [/] [100]

[else] [Tasa] [:=] [Base]

Análisis Léxico
Ejemplo

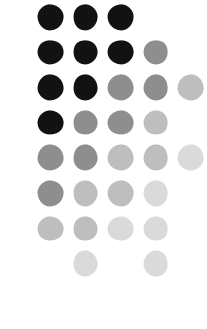


[59] [27] [80] [30]

[then**] [**Tasa**] [**:=**] [**Base**] [**+**] [**Recargo**] [**/**] [**100**]**

[else**] [**Tasa**] [**:=**] [**Base**]**

Análisis Léxico
Ejemplo

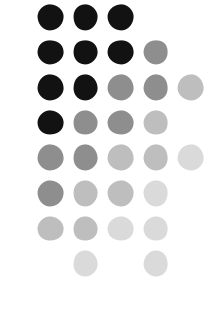


[59] [27] [80] [28]

[then**] [**Tasa**] [**:=**] [**Base**] [**+**] [**Recargo**] [**/**] [**100**]**

[else**] [**Tasa**] [**:=**] [**Base**]**

Análisis Léxico
Ejemplo

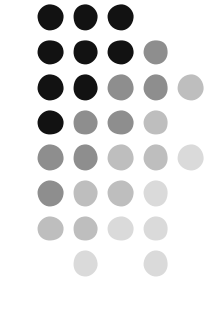


[59] [27] [80] [28]

[60] [27] [85] [27] [70] [27] [73] [28]

[61] [27] [85] [27]

Análisis Léxico
Ejemplo

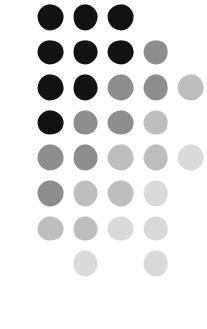


[59] [27, ‘Plazo’] [80] [28, ‘30’]

[60] [27, ‘Tasa’] [85] [27, ‘Base’] [70] [27,’Recargo’] [73] [28, ‘100’]

[61] [27,’Tasa’] [85] [27,‘Base’]

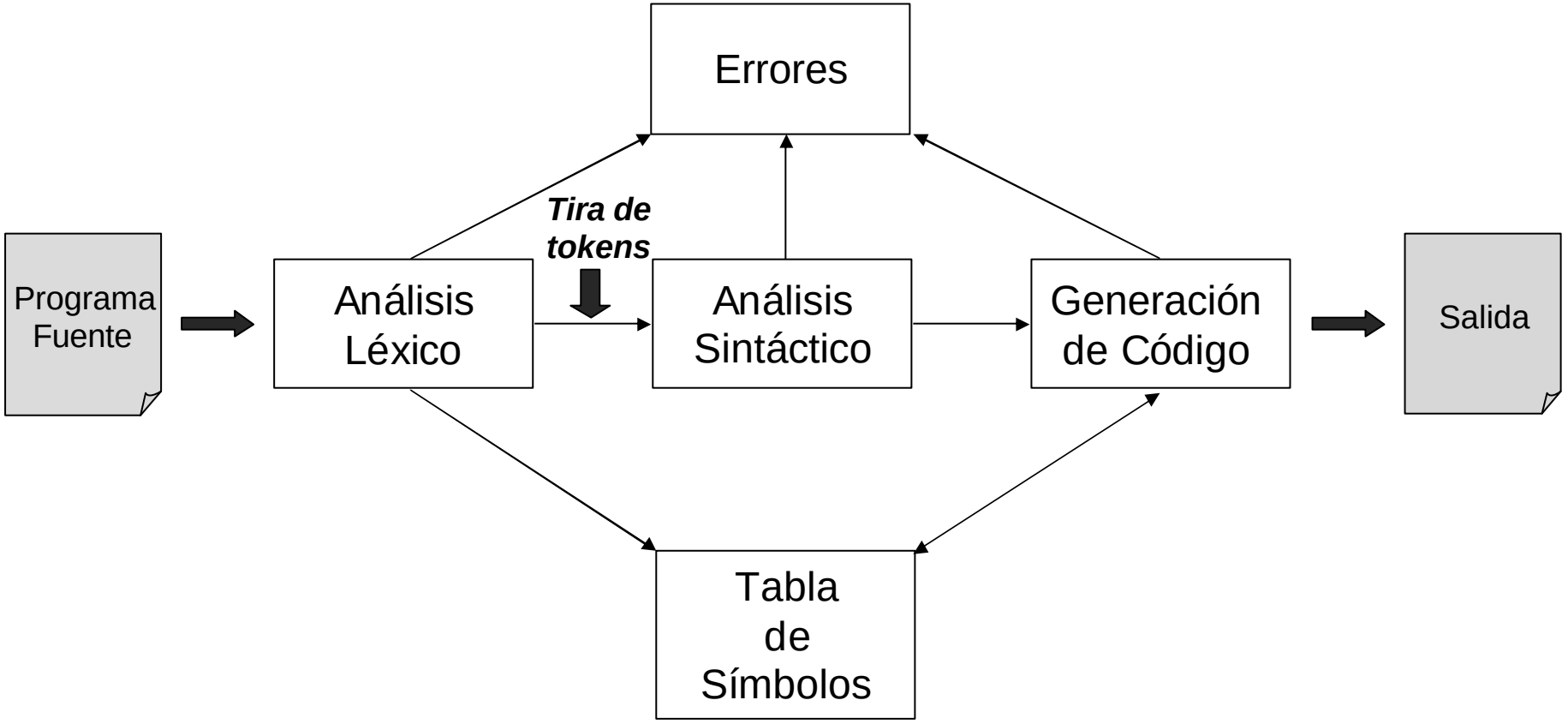
Análisis Léxico
Ejemplo



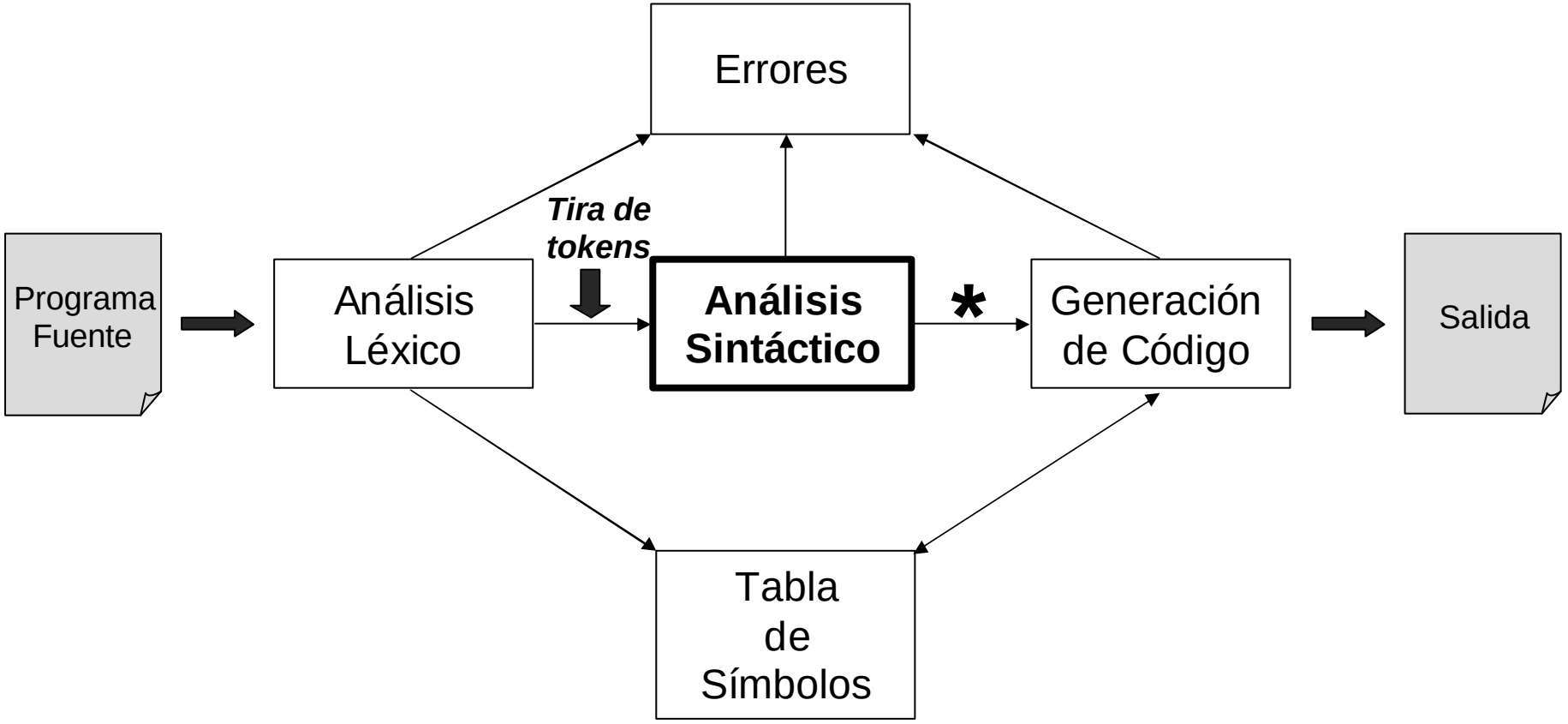
[59] [27] [80] [28] [60] [27] [85] [27] [70] [27] [73] [28] [61] [27] [85] [27]

IF ID >= CTE THEN ID := ID + ID / CTE ELSE ID := ID

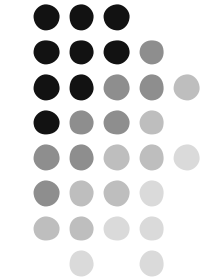
Fases de la Compilación



Fases de la Compilación

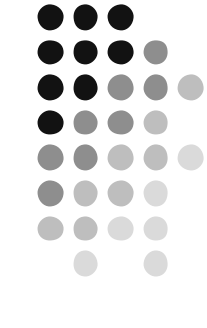


Análisis Sintáctico



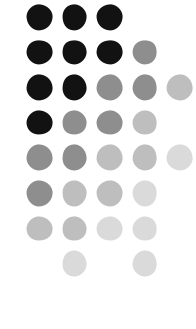
Agrupa los tokens del programa fuente en frases gramaticales que el compilador usará en las siguientes etapas.

Análisis Sintáctico



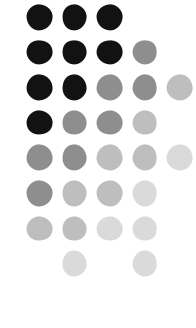
Los ***tokens*** son símbolos terminales en la ***gramática*** que describe al lenguaje fuente

Análisis Sintáctico



- La estructura jerárquica de un programa es representada por reglas que constituyen una ***gramática***.
- Las reglas se representan por medio de ***producciones***.
- Cada producción define un ***símbolo no terminal*** en función de ***símbolos terminales*** o tokens, y otros símbolos no terminales.
- Existe una producción que define al no terminal ***programa***.

Análisis Sintáctico
Gramática

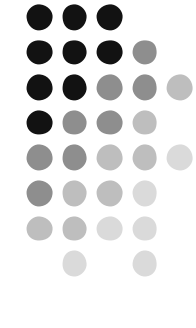


…

5. <sent> → <sel>
6. <sent> → <asig>
7. <sel> → IF <cond> THEN <sent> ELSE <sent>
8. <cond> → <exp> <comp> <exp>
9. <comp> → < | > | <= | >= | == | <>
10. <asig> → ID := <exp>
11. <exp> → <exp> + <term>
12. <exp> → <exp> - <term>
13. <exp> → <term>
14. <term> → <term> * <fact>
15. <term> → <term> / <fact>
16. <term> → <fact>
17. <fact> → ID
18. <fact> → CTE

…

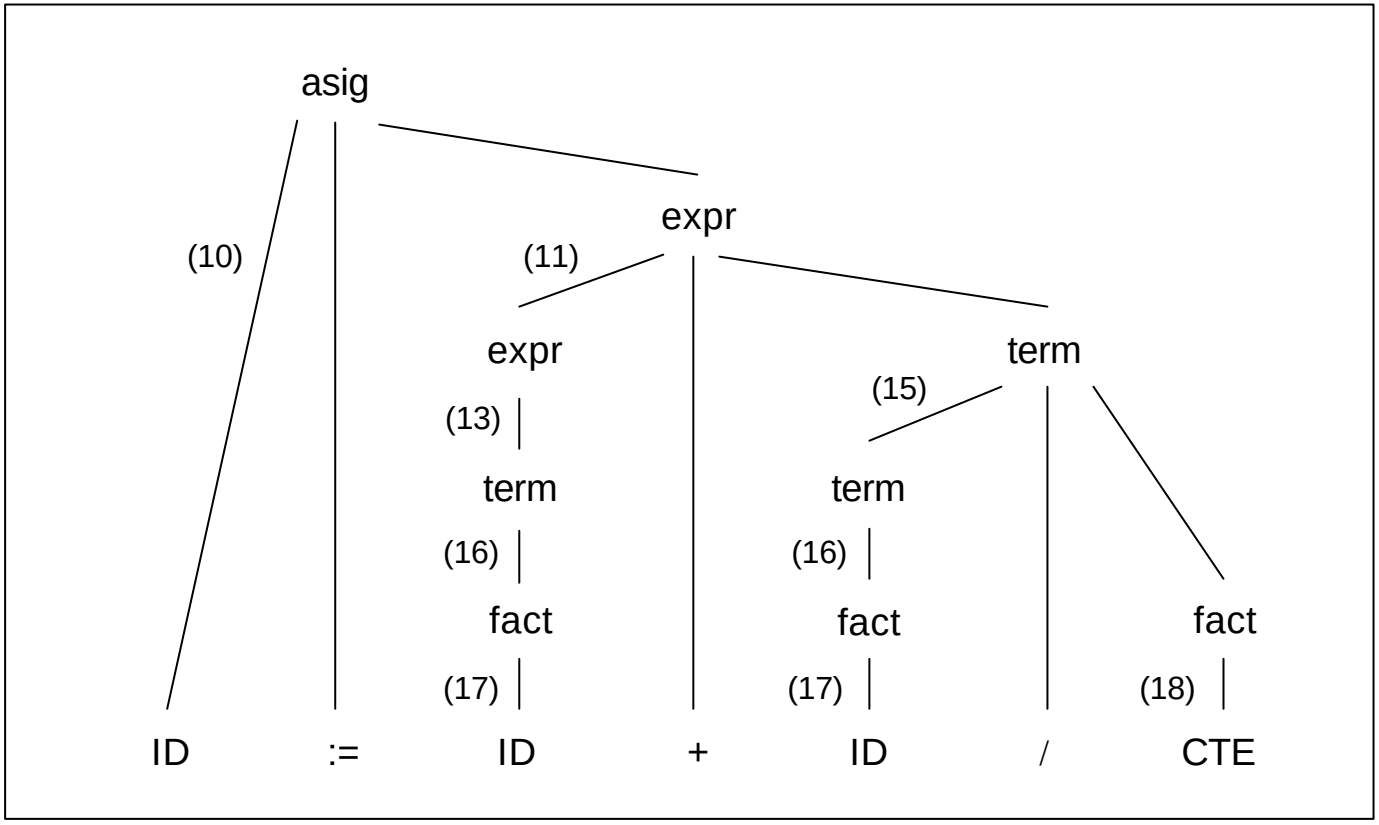
Análisis Sintáctico



- Usualmente, la estructura gramatical que el Análisis Sintáctico detecta en el código fuente es representada por un ***árbol de parsing***.
- El árbol de parsing demuestra como la secuencia de tokens de entrada puede ser derivada a partir de las reglas de una gramática.

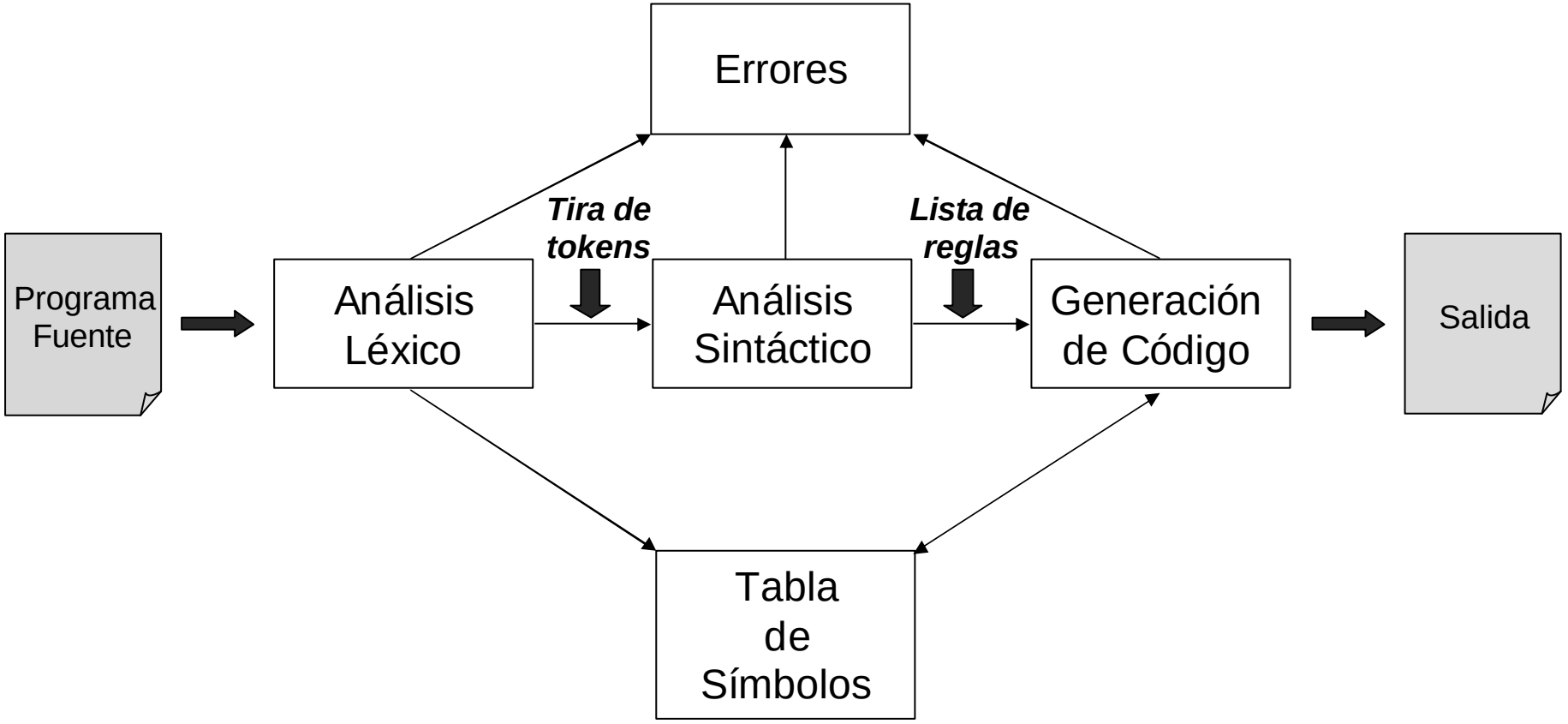
Análisis Sintáctico
Árbol de Parsing

Tasa := Base + Recargo / 100 ® ID := ID + ID / CTE

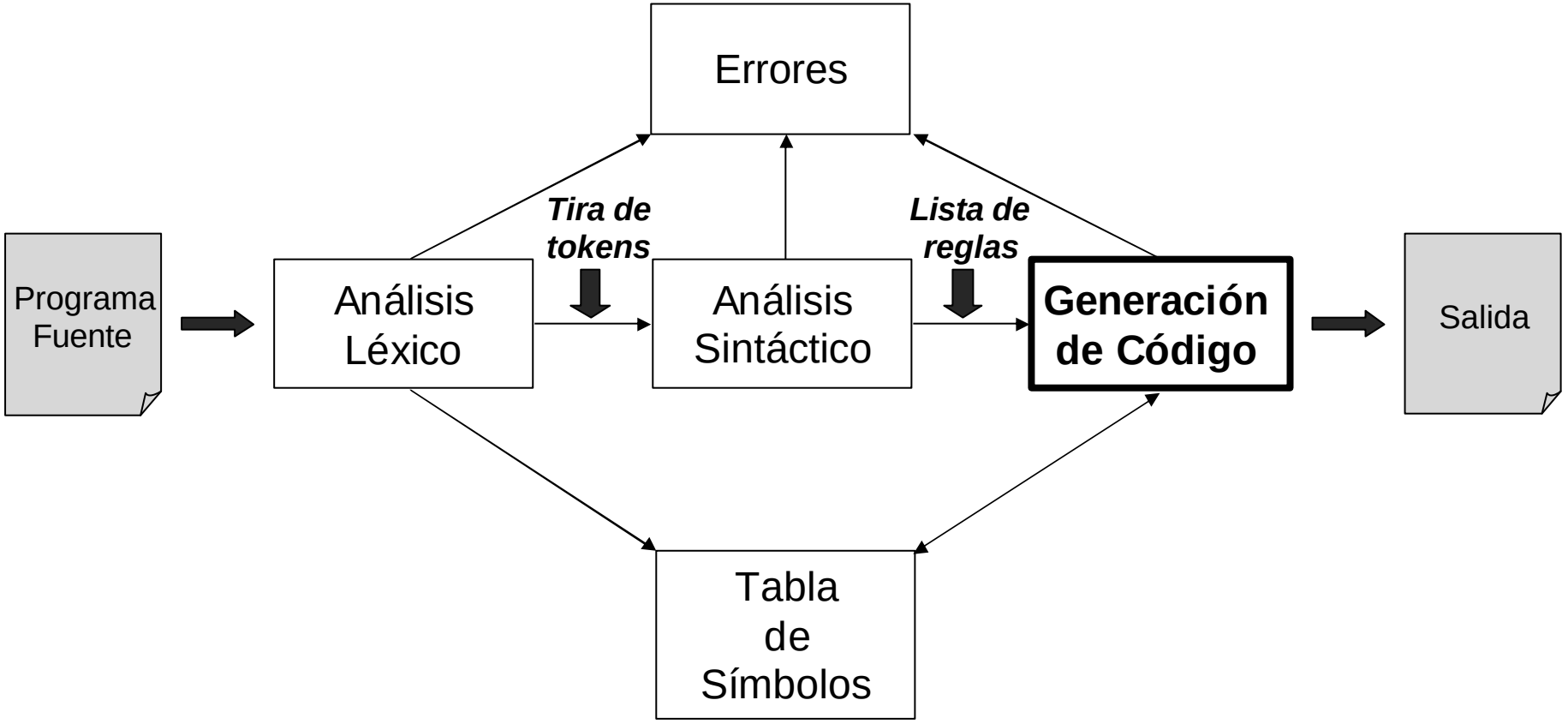


Lista de reglas: 17 16 13 17 16 18 15 11 10

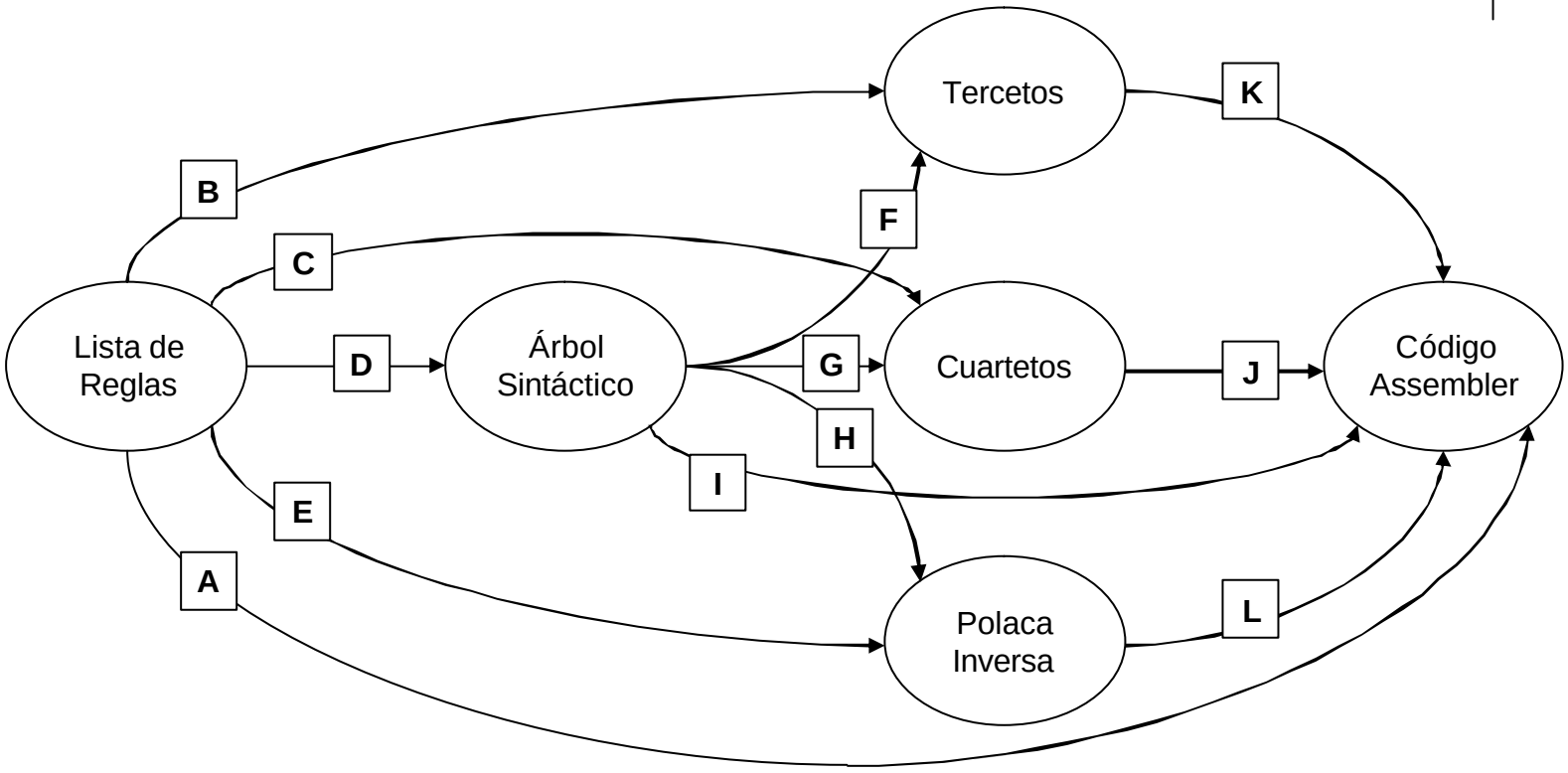
Fases de la Compilación



Fases de la Compilación



Generación de Código

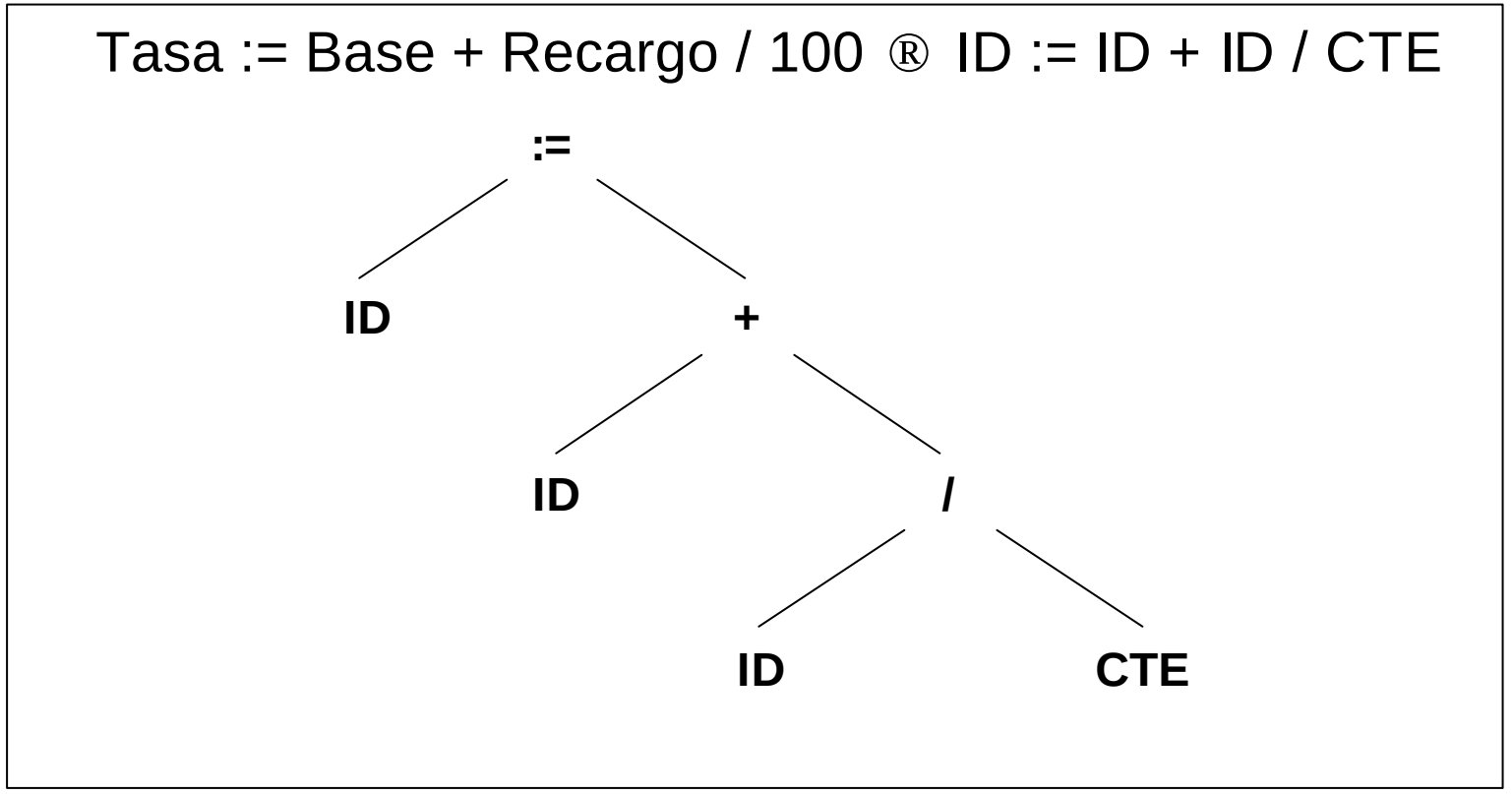


Caminos posibles:

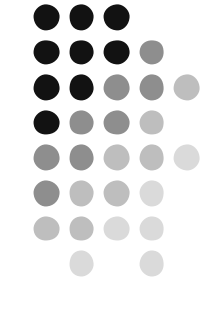
Camino 1: A
Camino 2: D, I
Camino 3: E, L
Camino 4: C, J
Camino 5: B, K
Camino 6: D, F, K
Camino 7: D, G, J
Camino 8: D, H, L

Generación de Código
Árbol Sintáctico

Es una representación comprimida del Árbol de Parsing.



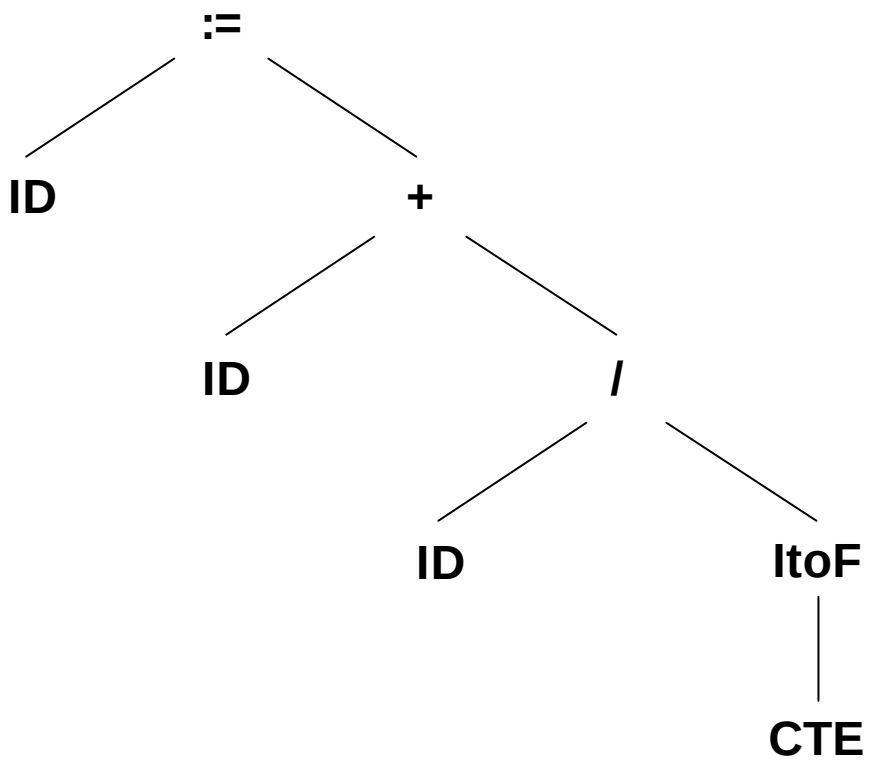
Análisis Semántico



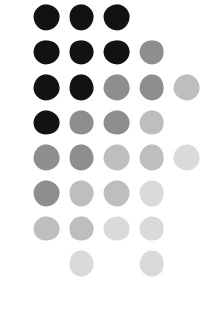
- Analiza el significado del programa.
- Chequea reglas que no pueden ser capturadas por la gramática, pero que pueden ser verificadas en tiempo de compilación. Estas reglas corresponden a la semántica estática del lenguaje.
- Ejemplos:
  - Chequeo de tipos en expresiones aritméticas.
  - Chequeo de tipo y número de parámetros en la llamada a una rutina.

Análisis Semántico
Ejemplo

Tasa := Base + Recargo / 100 ® ID := ID + ID / CTE



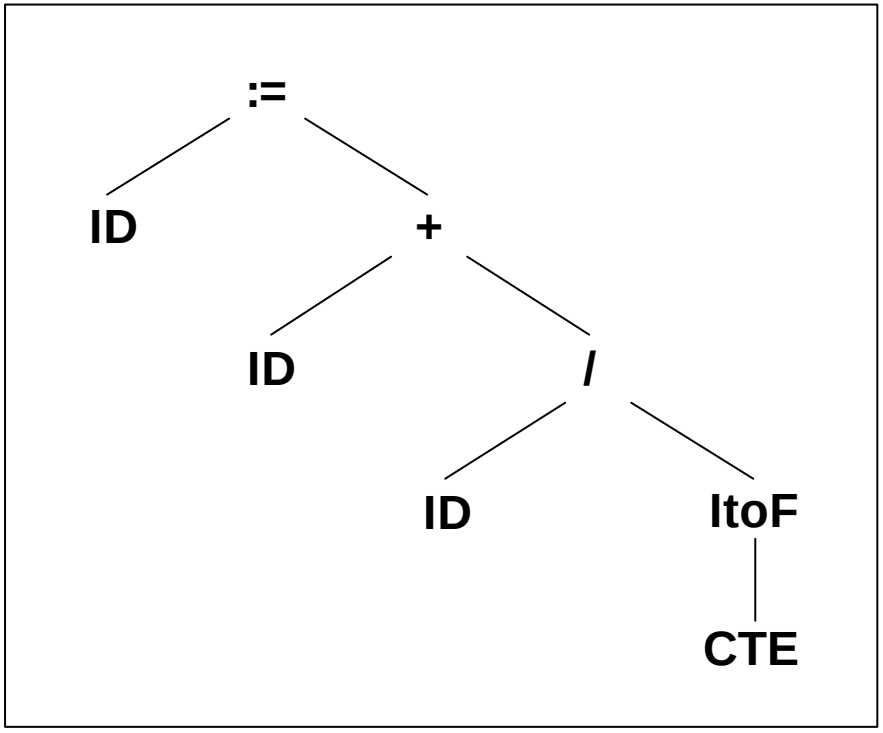
Código Intermedio



- Representación del código fuente como un programa escrito para ser ejecutado en una máquina abstracta.
- Posibles representaciones intermedias:
  - Tercetos
  - Cuartetos
  - Polaca Inversa

Código Intermedio
Ejemplo

Tasa := Base + Recargo / 100 ® ID := ID + ID / CTE

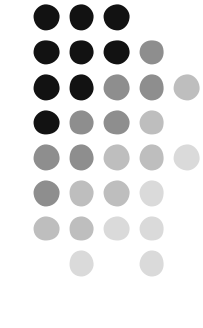


Árbol Sintáctico

14. …
15. (ItoF, 100, -)
16. (/, Recargo, [15])
17. (+, Base, [16])
18. (:=, Tasa, [17])
19. …

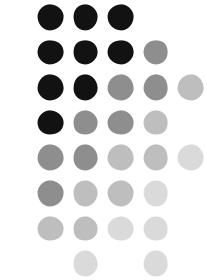
Tercetos

Optimización



- Transforma la representación actual del código en una nueva versión que logra el mismo resultado más eficientemente.
- Pueden aplicarse optimizaciones en diferentes etapas de la compilación:
  - durante la creación de la representación intermedia,
  - durante la transformación de una representación intermedia en otra,
  - durante la traducción del código intermedio a la salida,
  - luego de generar la salida,
  - e incluso durante la linkedición o la ejecución.

Optimización
Ejemplo



Tasa := Base + Recargo / 100 ® ID := ID + ID / CTE

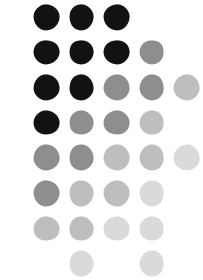
14. …
15. (ItoF, 100, -)
16. (/, Recargo, [15])
17. (+, Base, [16])
18. (:=, Tasa, [17])
19. …

Tercetos

14. …
15. (/, Recargo, 100.0)
16. (+, Base, [15])
17. (:=, Tasa, [16])
18. …

Tercetos Optimizados

Generación de Código propiamente dicho



- Se traduce la representación intermedia del programa fuente en el código nativo de la máquina destino.
- El código generado efectuará el chequeo de las reglas de semántica dinámica del lenguaje, que no pudieron ser verificadas durante la compilación.

Generación de Código Assembler
Ejemplo

Tasa := Base + Recargo / 100 ® ID := ID + ID / CTE

14. …
15. (/, Recargo, 100.0)
16. (+, Base, [15])
17. (:=, Tasa, [16])
18. …

Tercetos Optimizados

```
…
FLD, Recargo
FLD, Cte1
FDIV
FLD, Base
FADD
FSTP, Tasa
…
```

Código Assembler

Fases de la Compilación

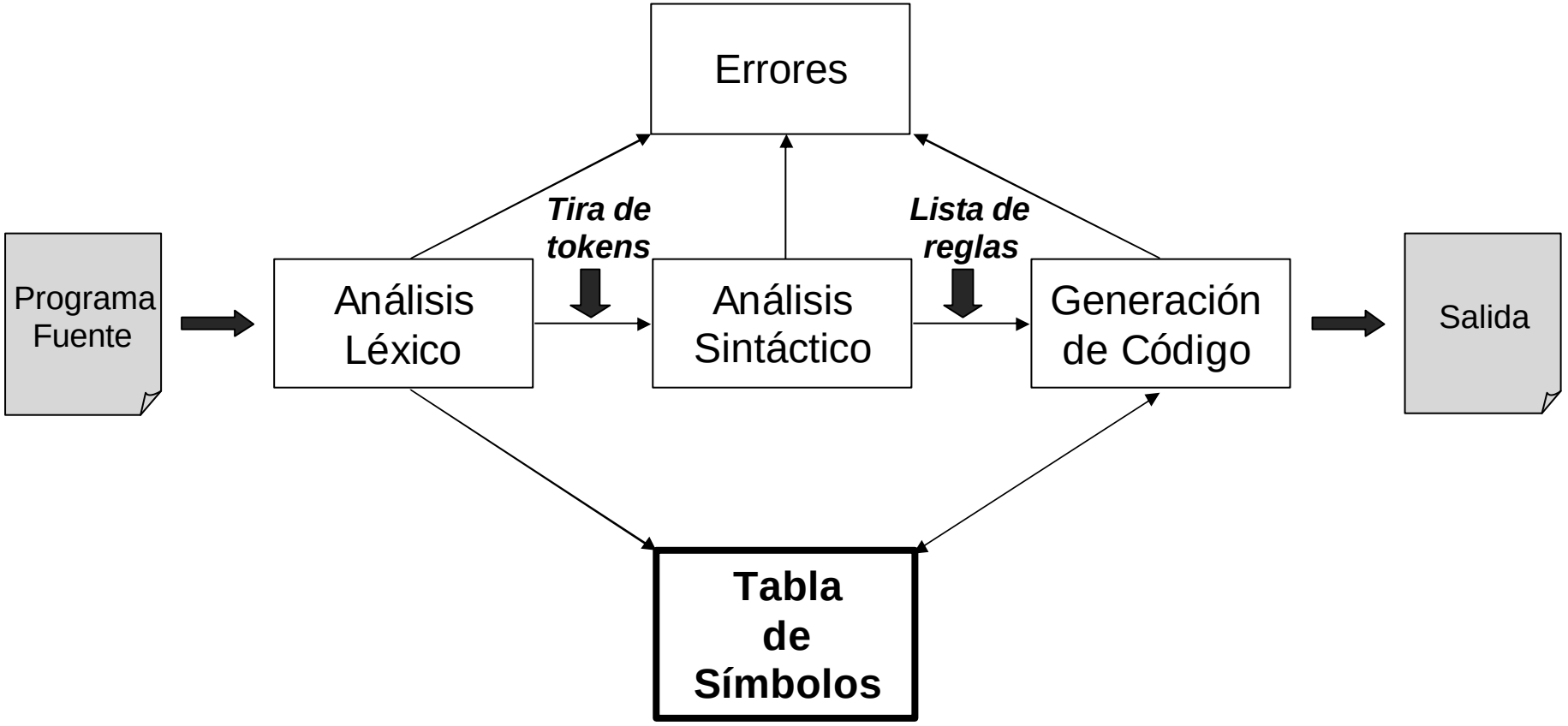
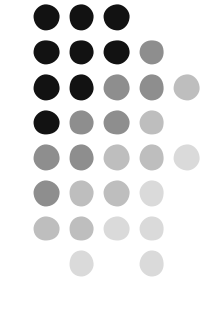
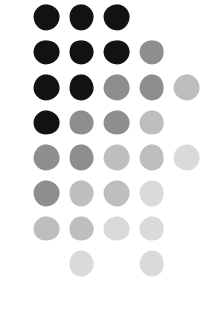


Tabla de Símbolos



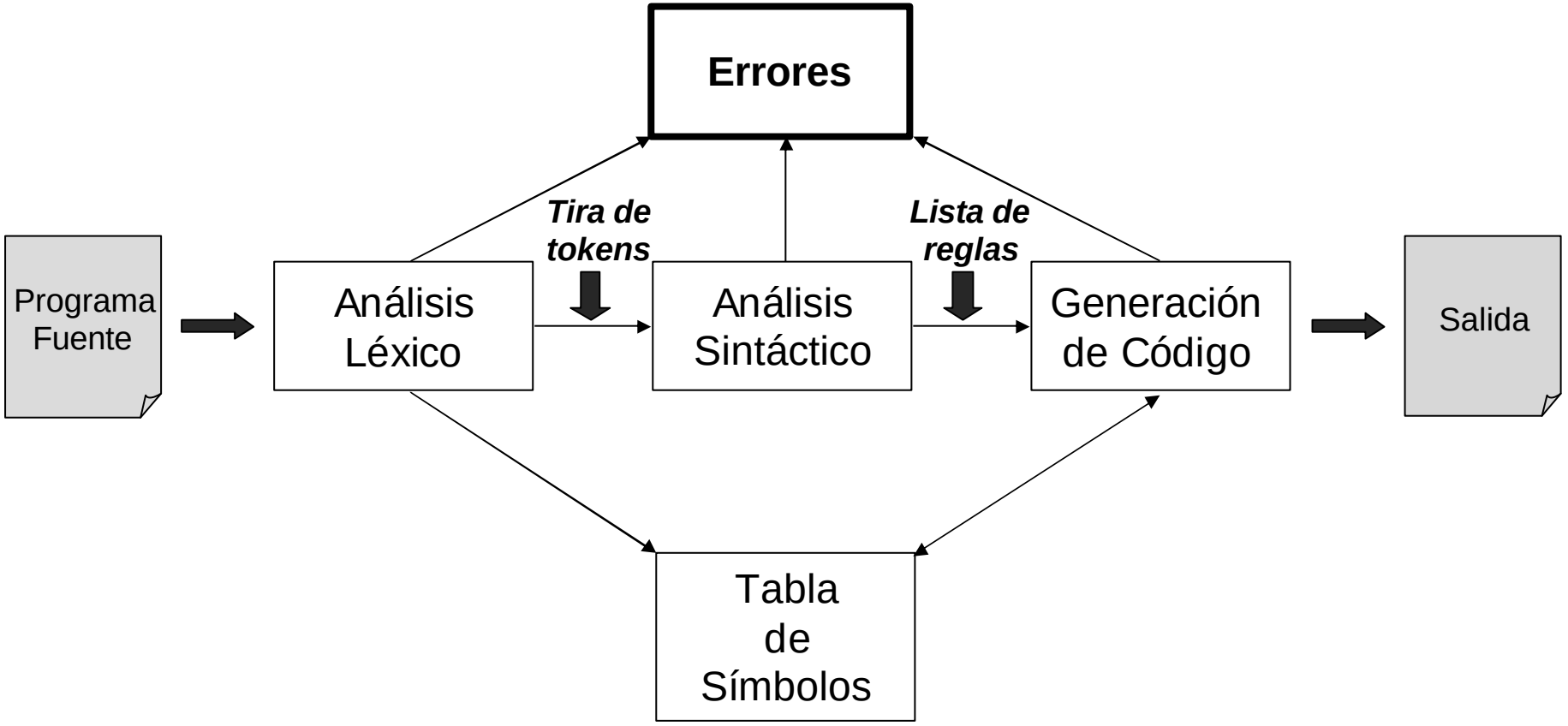
Es una estructura de datos que contiene un registro para cada identificador utilizado en el código fuente, con campos que contienen información relevante para cada símbolo (atributos).

Tabla de Símbolos

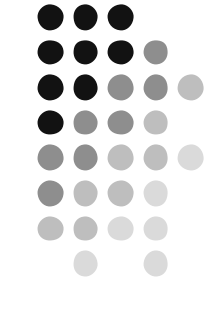


- Cuando el Análisis Léxico detecta un token de tipo identificador, lo ingresa en la Tabla de Símbolos.
- Durante la Generación de Código se ingresa información para los atributos de los símbolos, y se usa esa información de diversas maneras.
- Durante la Generación de Código puede ser necesario incorporar nuevas entradas a la Tabla de Símbolos.

Fases de la Compilación



Manejo de Errores



- Cada una de las etapas del Compilador puede detectar errores que son informados al programador.
- Un buen compilador no debería terminar su ejecución al detectar un error, sino que debería recuperarse y continuar con la compilación.