## Índice

*Marina de la Cruz*
*Alfonso Ortega*

# ¿Qué es un compilador?

- Un **compilador** es un programa que convierte el código fuente escrito en un lenguaje de programación de alto nivel a otra forma, normalmente código máquina.

```
unsigned int gcd (unsigned int a, unsigned int b)
{
    if (a == 0 &&b == 0)
        b = 1;
    else if (b == 0)
        b = a;
    else if (a != 0)
        while (a != b)
            if (a <b)
                b -= a;
            else
                a -= b;

    return b;
}
```

# Relación con la asignatura "Autómatas y Lenguajes"

- Como puede consultarse en las guías docentes, las asignaturas "Autómatas y Lenguajes" y "Proyecto de Autómatas y Lenguajes" están muy relacionadas.
- En el Proyecto, el objetivo es aprender las técnicas necesarias para poder construir procesadores de lenguaje (compiladores e intérpretes) y **desarrollar un compilador en el laboratorio.**
- En la asignatura "Autómatas y Lenguajes" el alumno aprende los fundamentos teóricos de los compiladores, y en las prácticas de dicha asignatura aprende un conjunto de herramientas de apoyo para el desarrollo de compiladores.

# Evolución de los lenguajes de programación

- Los lenguajes de programación han evolucionado con la potencia de los equipos:
  - Inicialmente sólo se podían programar **cableando físicamente** el programa en los circuitos del computador.
  - Posteriormente se facilitó la posibilidad de que los programas pudieran ser leídos desde el exterior añadiendo a los ordenadores la capacidad de configurar sus circuitos de manera adecuada, pero los programas seguían representando la **secuencia de 0s y 1s** que correspondía a que los diferentes circuitos estuvieran o no conectados. En esta época se utilizaban tarjetas y cintas perforadas.
  - Más adelante se facilitó el desarrollo de los programas distinguiendo en la secuencia de ceros y unos diferentes componentes (fundamentalmente operaciones y datos sobre las que realizarlas) y asociando nombres nemotécnicos a cada uno de ellos (por ejemplo MOV para mover un dato de una posición a otra ADD para sumar, etc.) De esta manera se pretendía minimizar los errores muy propensos por una representación tan poco intuitiva como la binaria. Aparecen de esta manera los **lenguajes simbólicos** que requieren de unos programas muy sencillos que traducen cada nombre nemotécnico a su secuencia de ceros y unos (ensambladores)
  - Desde este punto, el objetivo era expresar los algoritmos de la manera más cercana a la diseñada por el programador. Se estructuraron las diferentes alternativas para controlar el flujo de programa (if-then-else, while, etc.) y se intentó ofrecer esas construcciones de más alto nivel como manera de expresar los programas. Aparecen los **lenguajes de alto nivel**.
  - Surgieron también **paradigmas de programación** distintos de la estructurada que ofrecían ventajas a los programadores (programación funcional, lógica, orientada a objetos)
  - También se diseñaron herramientas de ingeniería del software que pretendían automatizar en la medida de lo posible la programación de aplicaciones.

## Introducción

- Los programas escritos en lenguaje máquina son los únicos que se pueden ejecutar directamente en una computadora. **Los restantes hay que traducirlos**.

- Los lenguajes simbólicos se traducen mediante programas llamados **ensambladores**, que convierten cada instrucción simbólica en la instrucción máquina equivalente. Estos programas suelen ser relativamente sencillos y no se van a considerar aquí.

- Los programas escritos en lenguajes de alto nivel se traducen mediante programas llamados, en general, **traductores o procesadores de lenguaje**. Existen dos tipos de estos traductores:
  - Compiladores
  - Intérpretes

## Compiladores e intérpretes

- Un compilador analiza un programa escrito en un lenguaje de alto nivel (programa fuente) y, si es correcto, genera un código equivalente (programa objeto) escrito en otro lenguaje, que puede ser de primera generación (de la máquina), de segunda generación (simbólico), o de tercera generación. El programa objeto puede guardarse y usarse tantas veces como se quiera, sin necesidad de traducirlo de nuevo.

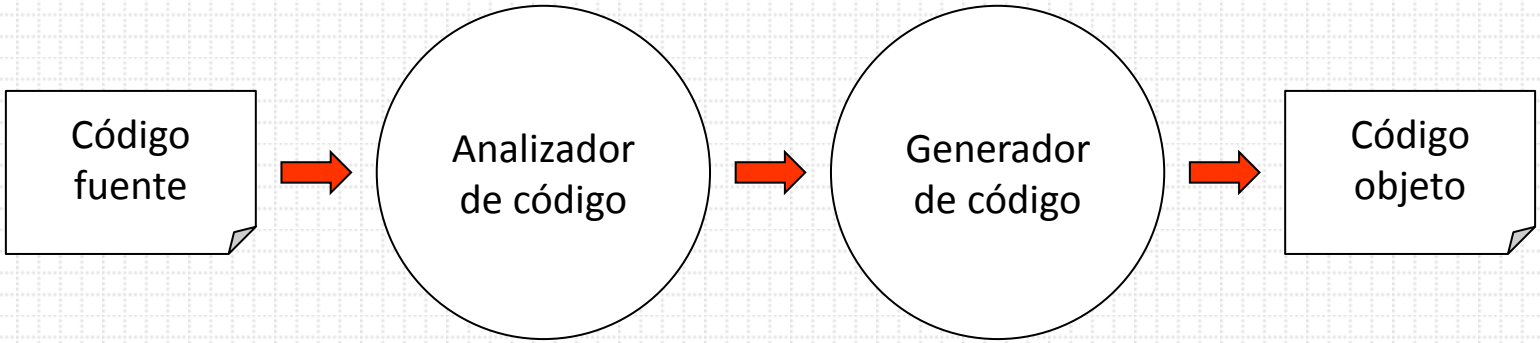
- Algunos compiladores: C, C++, C#.

- Un intérprete analiza un programa escrito en un lenguaje de alto nivel y, si es correcto, lo ejecuta directamente en el lenguaje de la máquina en que se está ejecutando el intérprete. Cada vez que se desea ejecutar el programa, es preciso interpretarlo de nuevo.

- Algunos intérpretes: Lisp, Prolog, Python, JavaScript.

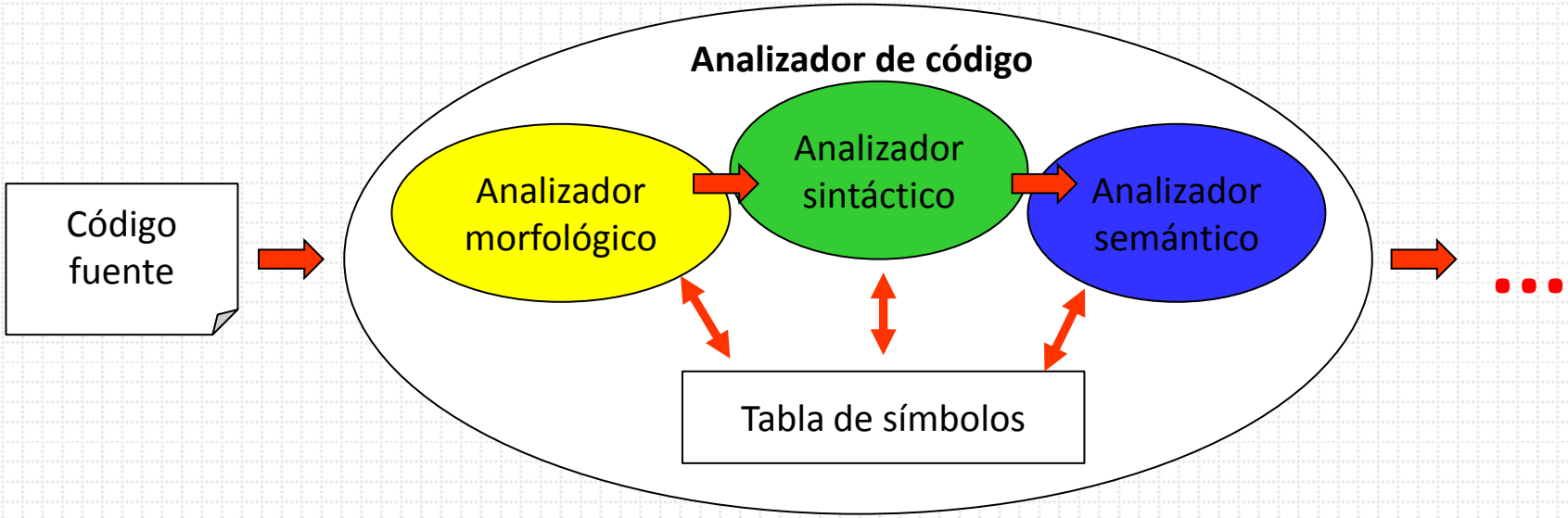
## Partes de un compilador

- Se pueden distinguir dos grandes componentes en un compilador:
  - **El analizador de código fuente**: toma como entrada el programa con el código fuente (esto es, una secuencia de caracteres), y lo interpreta como una estructura de símbolos (variables, constantes, operadores, etc.).
  - **El generador de código objeto**: toma como entrada el resultado del analizador de código fuente y produce código objeto como salida.



## El analizador de código

- El analizador de código tiene normalmente tres etapas:
  - **Analizador morfológico**: transforma el código fuente del programa de una secuencia de caracteres, a una secuencia de unidades sintácticas (tokens).
  - **Analizador sintáctico**: interpreta las unidades sintácticas identificadas (tokens) por el analizador morfológico como un programa estructurado según una gramática (ya se verá en Autómatas y Lenguajes).
  - **Analizador semántico**: realiza comprobaciones semánticas como por ejemplo que las variables están declaradas antes de su uso, que los tipos de las expresiones son correctos, etc.



## El analizador morfológico (I)

- También se le llama analizador léxico o scanner.
- Se hace cargo del análisis más sencillo.
- **Transforma** el código fuente en una secuencia de unidades sintácticas (tokens):
  - Identificadores
  - Palabras reservadas (begin, for, if, end,…)
  - Constantes numéricas (enteras, reales, etc...)
  - Constantes literales o de tipo "string".
  - Símbolos simples (operadores, +, -, *,..., separadores, ;, ., ...)
  - Símbolos múltiples (operadores, +=, -=, >=, <=,...)
- Esta transformación permite que el analizador sintáctico no vea el fichero como una secuencia de caracteres sino como una secuencia de elementos con significado sintáctico (tokens).

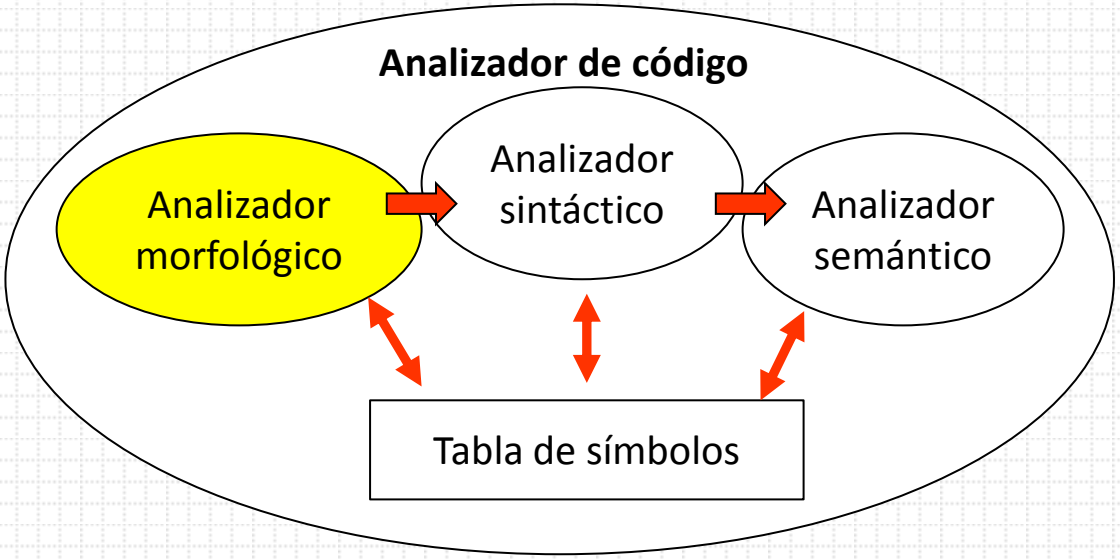
## El analizador morfológico (II)

- Además, **ignora** las partes del código fuente no compilables:
  - Comentarios
  - Espacios en blanco (blancos, tabuladores y saltos de línea)

- Detecta **errores morfológicos**:
  - Símbolos no permitidos
  - Identificadores mal construidos
  - Constantes mal construidas
  - Comentarios mal construidos

- Puede acceder a la **tabla de símbolos** si es necesario (depende del diseño del compilador).

## El analizador morfológico (III)




```
begin
    int A;

    A := 100;
    A := A+A;
    output A
end
```

**A M**

```
(<palabra clave>,begin)
    (<palabra clave>,int)(<id>,A)(<simb>,;)
    (<id>,A)(<simbm>,:=)
        (<cons int>,100)(<simb>,;)
    (<id>,A)(<simbm>,:=)
        (<id>,A)(<simb>,+)(<id>,A)(<simb>,;)
    (<palabra clave>,output)(<id>,A)
(palabra clave>,end)
```

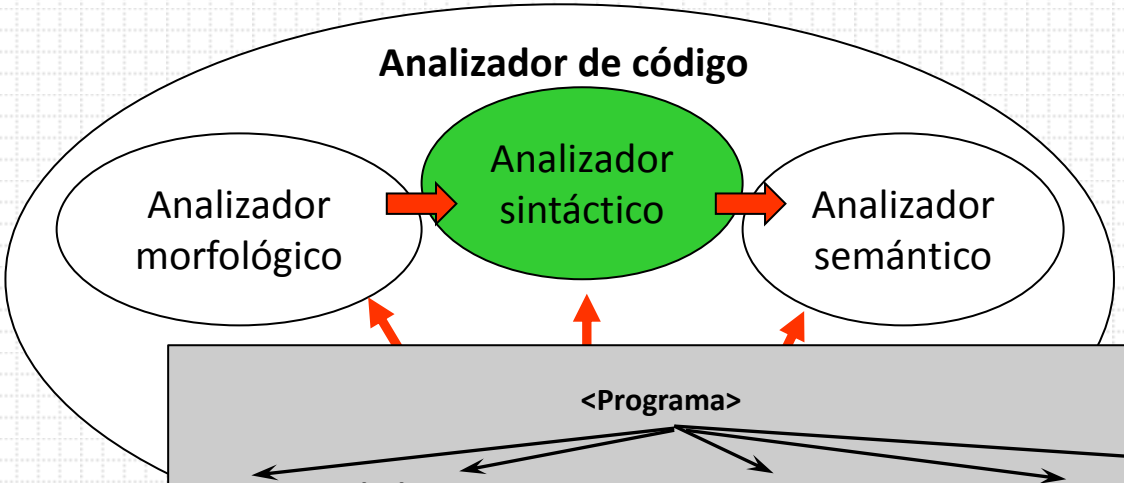
## El analizador sintáctico (I)

- También se le denomina parser.
- Tiene como **entrada** la secuencia de tokens que construye el analizador morfológico.
- **Comprueba,** a partir de la gramática independiente del contexto, que las unidades sintácticas identificadas por el analizador morfológico mantienen una estructura sintácticamente correcta (de acuerdo al lenguaje para el que está diseñado el compilador).
- **Genera** una representación estructural del programa, mostrando por ejemplo, qué unidades sintácticas se agrupan en una sentencia, qué sentencias se agrupan en bloques, etc. Esta estructura se llama **árbol de derivación**.

## El analizador sintáctico (II)

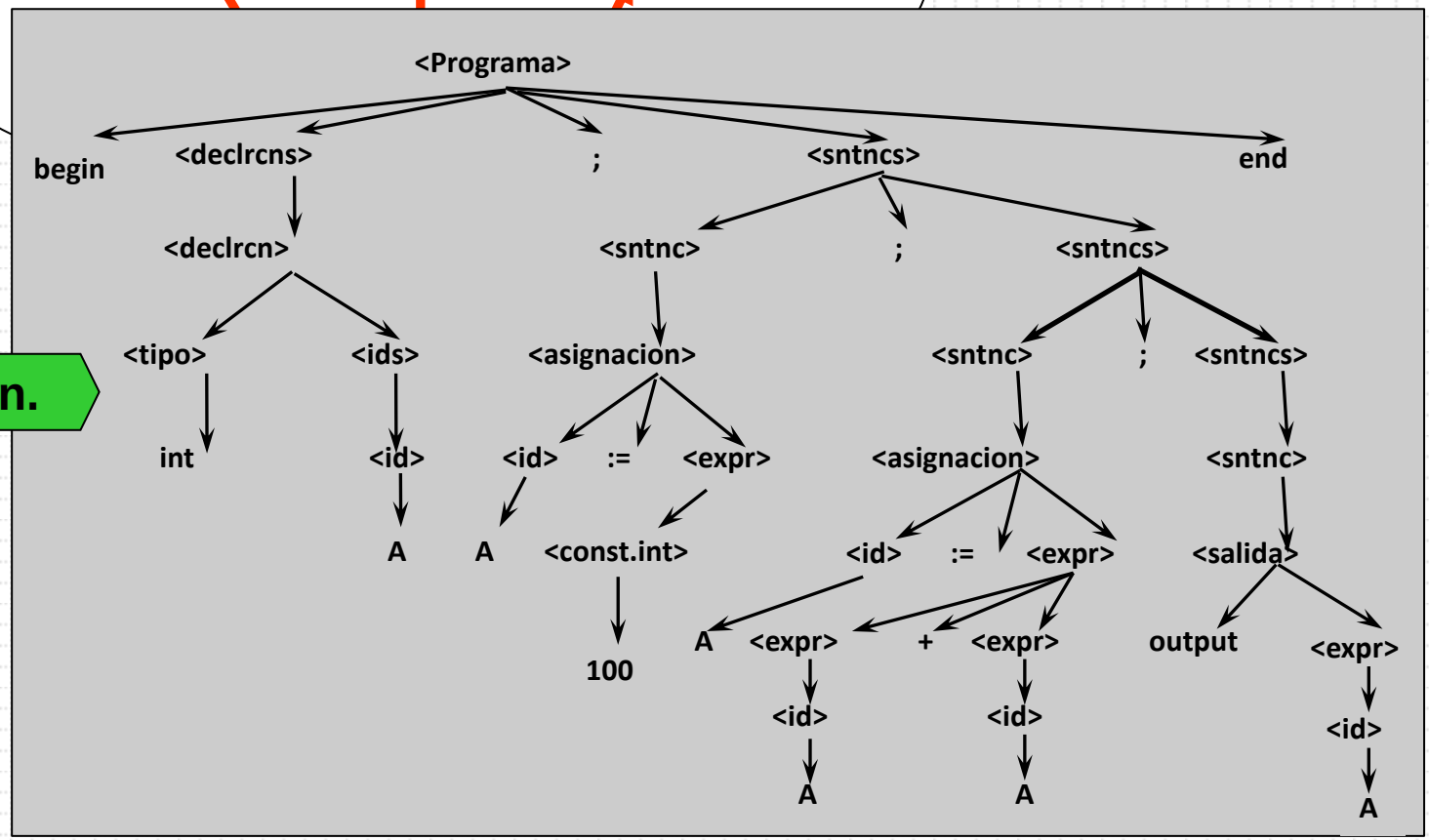
- El analizador sintáctico también se encarga de detectar los **errores sintácticos** que pudiera haber en el código:
  - Encontrando los lugares del código en las que las secuencias de unidades sintácticas no cumplen las reglas sintácticas del lenguaje.
  - Por ejemplo: falta de ';' al final de una sentencia Java.
- Puede acceder a la **tabla de símbolos** si es necesario.
- En algunos compiladores, el analizador sintáctico es el **programa principal** del compilador y utiliza el resto de las partes (analizador morfológico, analizador semántico, generador de código, etc) como subrutinas.

## El analizador sintáctico (III)




```
(<palabra clave>,begin)
   (<palabra clave>,int)(<id>,A)(<simb>,:)
   (<id>,A)(<simbm>,:=)
      (<cons int>,100)(<simb>,;)
   (<id>,A)(<simbm>,:=)
      (<id>,A)(<simb>,+)(<id>,A)(<simb>,;)
   (<palabra clave>,output)(<id>,A)
(palabra clave>,end)
```

**A Sin.**



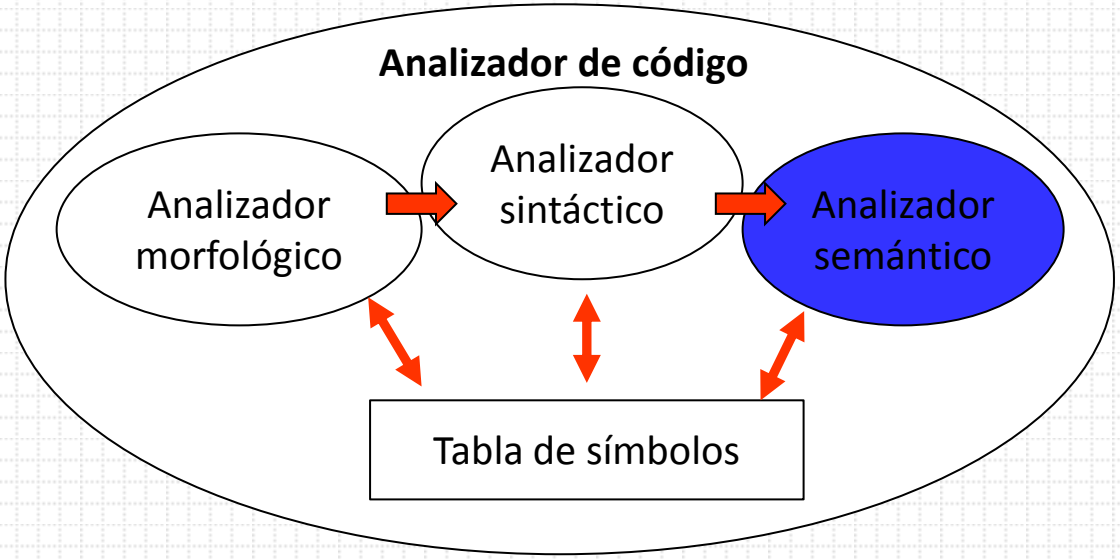
## Tabla de símbolos (I)

- La tabla de símbolos, es la parte en la que se guarda la información necesaria para poder determinar si el programa es correcto y (si lo es) generar el código.

- Esta información se centra en los **identificadores** que aparecen en el programa fuente (variables, funciones, procedimientos, etiquetas, etc) y de cada uno de ellos suele guardarse:
  - Tipo del identificador
  - Ámbito de validez del identificador
  - Si el identificador corresponde a una función, el número y tipo de sus argumentos
  - Si el identificador corresponde a un vector, el tamaño del mismo
  - etc

- Suele utilizarse alguna **estructura de datos** (por ejemplo tablas hash) en la que se puede implementar de forma eficiente las operaciones:
  - Insertar información
  - Recuperar información

## Tabla de símbolos (II)




```
begin
    int A;

    A := 100;
    A := A+A;
    output A
end
```

A M

A Sin.

| *Elemento* | *Tipo* | *Valor* |
|---|---|---|
| <id> | int | A |

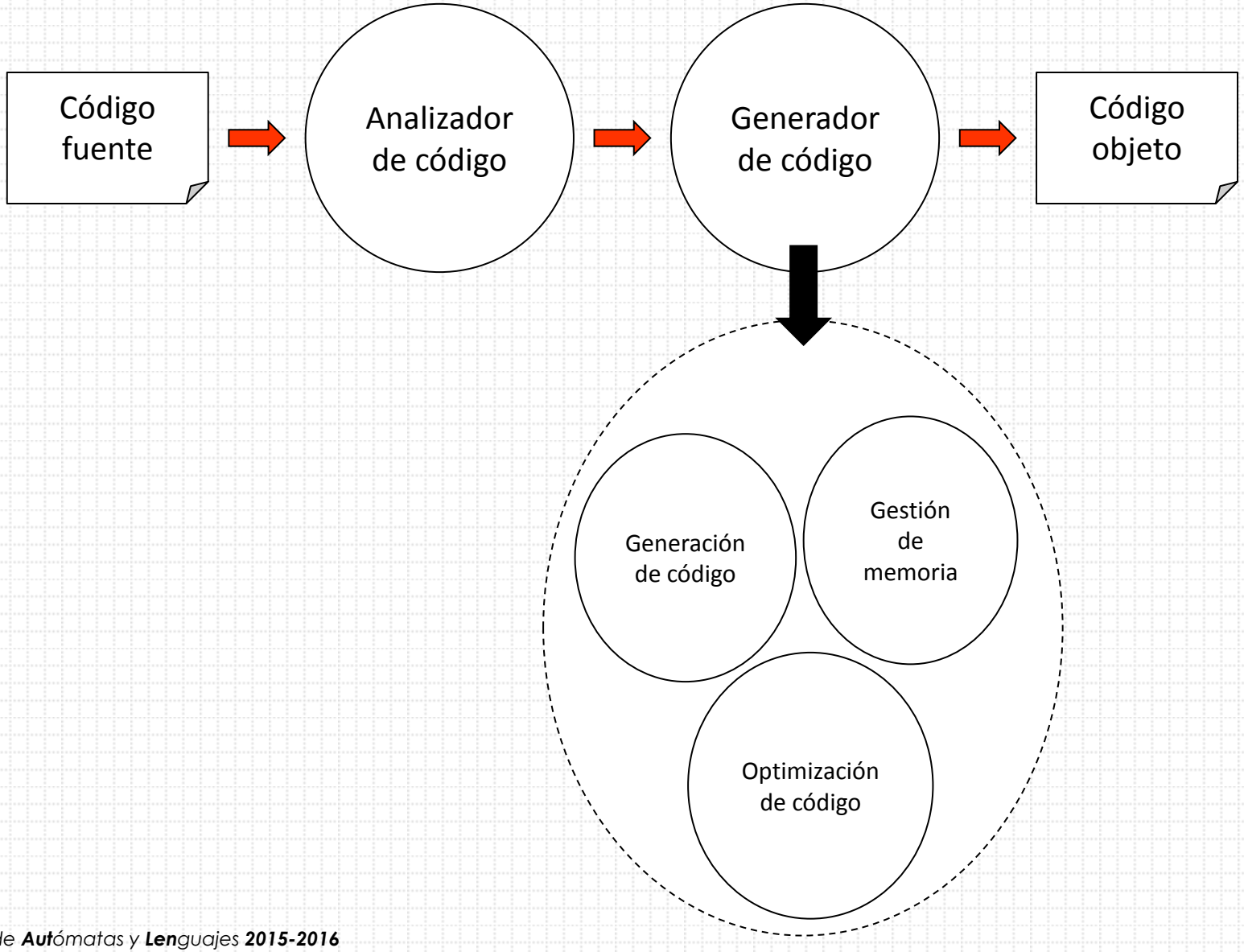
## El analizador semántico (I)

- Es la parte del compilador que realiza las tareas que completan el análisis sintáctico con la comprobación de la **corrección semántica** del programa fuente.

- Alguna de sus **tareas** puede ser:
  - Comprobar la unicidad de identificadores.
  - Comprobar la declaración de identificadores antes de su uso.
  - Comprobar la compatibilidad de tipos en expresiones y asignaciones.
  - Comprobar el tipo de los parámetros en las llamadas a funciones.

- Puede acceder a la tabla de símbolos si es necesario.

## El generador de código (I)



## El generador de código (II)

- **Generación de código**
  - Generación de las instrucciones del programa objeto "equivalente" al fuente.
  - En esta tarea se puede acceder a la tabla de símbolos si es necesario.

- **Gestión de memoria** y **optimización de código**:
  - Alguna de las diferencias más importantes entre la escritura directa "a mano" en el lenguaje objeto de un programa "equivalente" al programa fuente y la generación automática de ese programa mediante un traductor es que resulta difícil para los traductores hacer un uso tan **eficiente** de los recursos.
  - La gestión de memoria y la optimización de código intentan mitigar esas diferencias.

- Un compilador puede tener distintos módulos de generación de código para generar código para distintas plataformas.

## El generador de código (III)

- El traductor podría haber generado inicialmente la siguiente versión del programa en NASM.
- El optimizador podría darse cuenta de que, en este caso, la gestión de expresiones aritméticas realiza dos sentencias que resultan redundantes que se pueden eliminar.

```
begin
    int A;

    A := 100;
    A := A+A;
    output A
end
```

**Generación Código**

```
segment .data
        _A dd 0
segment .codigo
        global _main
_main:
        push dword 100
        pop eax
        mov  [_A], eax
        push dword [_A]
        push dword [_A]
        POP edx
        POP eax
        add eax,edx
        push eax
        pop eax
        mov  [_A], eax
        push dword [_A]
        pop eax
        push eax
        call imprime_entero
        add esp, 4
        call imprime_fin_linea
        ret
```

## El generador de código (IV)

- Para obtener la siguiente versión equivalente y más sencilla.

**Optimización y G. Memoria**

```
segment .data
        _A dd 0
segment .codigo
        global _main
_main:
        push dword 100
        pop eax
        mov  [_A], eax

        push dword [_A]
        POP edx

        add eax,edx
        push eax
        pop eax
        mov  [_A], eax
        push dword [_A]
        pop eax
        push eax
        call imprime_entero
        add esp, 4
        call imprime_fin_linea
        ret
```

## Resumen

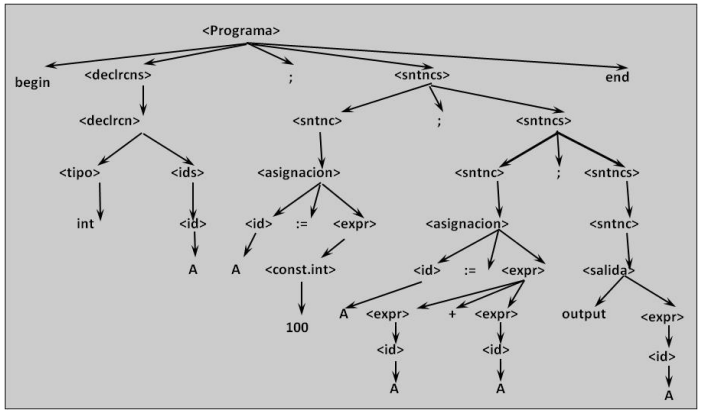
### Fuente

```
begin
   int A;

   A := 100;
   A := A+A;
   output A
end
```

### COMPILACIÓN

A M → A Sin. → A Sem. → Generación Código Optimización y G. Memoria




```
(<palabra clave>,begin)
   (<palabra clave>,int)(<id>,A)(<simb>,;)
   (<id>,A)(<simbm>,:=)
      (<cons int>,100)(<simb>,;)
   (<id>,A)(<simbm>,:=)
      (<id>,A)(<simb>,+)(<id>,A)(<simb>,;)
   (<palabra clave>,output)(<id>,A)
(palabra clave>,end)
```

| *Elemento* | *Tipo* | *Valor* |
|---|---|---|
| <id> | int | A |

### Objeto

```
segment .data
        _A dd 0
segment .codigo
        global _main
_main:
        push dword 100
        pop eax
        mov  [_A], eax

        push dword [_A]
        POP edx

        add eax,edx
        push eax
        pop eax
        mov  [_A], eax
        push dword [_A]
        pop eax
        push eax
        call imprime_entero
        add esp, 4
        call imprime_fin_linea
        ret
```

# Otros conceptos

- **Pasada:** Recorrido total de todo el fuente con alguna misión específica.

- **Compiladores de una pasada y de varias pasadas:**
  - Los de una pasada, realizan todo el trabajo con sólo un recorrido del fuente.
  - Los de varias pasadas realizan más recorridos y, por tanto, imponen menos restricciones al lenguaje fuente.

- Uno de los mayores servicios de los traductores es el diagnóstico de los **errores** cometidos en la codificación.

- Podemos distinguir dos conceptos:
  - **Detección de errores**: el compilador es capaz de detectar en el programa fuente errores de distinta naturaleza:
    - Morfológicos
    - Sintácticos
    - Semánticos
  - **Recuperación ante los errores**: el compilador, para poder detectar varios errores y no detenerse ante el primer error, necesita un mecanismo que le permita "recuperarse" ante un error en el programa fuente y poder continuar con la traducción del mismo.