Universidad
Europea de Madrid

LAUREATE INTERNATIONAL UNIVERSITIES

# INTRODUCCIÓN A COMPILADORES Y LENGUAJES FORMALES

## LENGUAJES FORMALES

© Todos los derechos de propiedad intelectual de esta obra pertenecen en exclusiva a la Universidad Europea de Madrid, S.L.U. Queda terminantemente prohibida la reproducción, puesta a disposición del público y en general cualquier otra forma de explotación de toda o parte de la misma.

La utilización no autorizada de esta obra, así como los perjuicios ocasionados en los derechos de propiedad intelectual e industrial de la Universidad Europea de Madrid, S.L.U., darán lugar al ejercicio de las acciones que legalmente le correspondan y, en su caso, a las responsabilidades que de dicho ejercicio se deriven.

# Índice

Presentación 4

¿Cuál fue la necesidad de los lenguajes formales? 5

Definiciones básicas 6

- Definición de los conceptos: esenciales y su notación 6

Operaciones con palabras 8

Operaciones con lenguajes 9

Gramáticas formales 11

Derivaciones y árbol de derivación 12

Ambigüedad 14

Eliminar la ambigüedad 15

- Eliminar la recursividad a izquierdas 15
- Factorizar por la izquierda 15

Clasificación de los lenguajes de programación 17

Ventajas e inconvenientes de los lenguajes de alto nivel 19

Resumen 20



© Universidad Europea de Madrid. Todos los derechos reservados.

## Presentación

El objetivo de este tema es que el estudiante comprenda la necesidad y el origen de los **lenguajes formales,** junto con la forma de **especificar una gramática y a reconocer una sentencia,** así como a **identificar y eliminar ambigüedades.**

El objetivo en este segundo tema es llegar a conocer los siguientes puntos:

- ¿Cuál fue la necesidad de los lenguajes formales?
- Definiciones básicas.
- Operaciones con palabras.
- Operaciones con lenguajes.
- Gramáticas formales.
- Derivaciones y árboles de derivación.
- Ambigüedad.
- Eliminación de la ambigüedad.
- Clasificación de los lenguajes de programación.
- Ventajas e inconvenientes de los lenguajes de alto nivel.

Al inicio veremos un pequeño repaso histórico para entender la motivación de los lenguajes formales, junto con la jerarquía de Chomsky, lo que permitió abrirse a otro enfoque y llegar a la solución que tenemos actualmente.



© Universidad Europea de Madrid. Todos los derechos reservados.

## ¿Cuál fue la necesidad de los lenguajes formales?

Allá por 1954, se inició el desarrollo del **lenguaje de programación FORTRAN** que, aunque fue un verdadero compilador, adolecía de una **gran complejidad.** Casi al mismo tiempo que John Backus inició su compilador, Noam Chomsky empezó con el estudio del **lenguaje natural,** con la idea de estructurarlo, y consiguió una revolución en este campo.

Una de sus muchas aportaciones fue la **jerarquía de Chomsky** para organizar las gramáticas formales, entendiéndose por gramática las reglas necesarias para definir la estructura de un lenguaje formal.

La jerarquía de Chomsky estructura las gramáticas en **cuatro niveles,** del más genérico y que incluye a los demás (tipo 0) al más específico (tipo 3):

| | |
|---|---|
| Gramáticas tipo 0 | Sin restricciones. Estas gramáticas tienen que tener en su parte izquierda al menos un símbolo no terminal (posteriormente veremos lo que significa, el concepto de **símbolo terminal y no terminal**). |
| Gramáticas tipo 1 | Dependientes del contexto. Se las denomina **dependientes del contexto** porque hay que tener en cuenta los símbolos que vienen antes y después del que queremos sustituir (su contexto). |
| Gramáticas tipo 2 | Independientes del contexto. Generan **lenguajes independientes del contexto** y se caracterizan porque en la parte izquierda de una producción **solo pueden tener un símbolo no terminal.** |
| Gramáticas tipo 3 | Expresiones regulares. Estas son las gramáticas más restrictivas y generan lenguajes regulares. En su parte izquierda tienen solo un no terminal y en su parte derecha tienen solo un terminal. |

A nosotros, como diseñadores de lenguajes formales, nos interesan las **gramáticas tipo 2,** que son las que nos permiten definir un lenguaje de programación, y las de **tipo 3**, que nos permitirán definir cuáles son los caracteres que constituyen las palabras de nuestro lenguaje.

Posteriormente, veremos un ejemplo de cada uno de estos cuatro tipos de gramática, una vez hayamos definido los conceptos necesarios para entenderlos.

Definiremos también a lo largo de esta Unidad la notación básica necesaria que utilizaremos para especificar un **lenguaje formal.**



© Universidad Europea de Madrid. Todos los derechos reservados.

## Definiciones básicas

Un lenguaje natural es el lenguaje hablado o escrito que tiene como misión principal establecer la comunicación. Ejemplo de estos lenguajes son: español, inglés, francés, chino, etc.

Podemos definir un **lenguaje formal,** además de como una especialización del lenguaje natural**,** como el conjunto de palabras que están formadas por caracteres o símbolos, de longitud finita, que a su vez forman parte de un alfabeto finito. Con los lenguajes formales construimos los lenguajes de programación, y ejemplo de ellos son:

- **C**
- **C++**
- **Java**
- **Etc**

### Definición de los conceptos: esenciales y su notación

| | |
|---|---|
| Carácter o símbolo | Es el componente indivisible y elemental con el que se compone un texto. Ejemplos: a, b, 5, [, ) |
| Alfabeto (∑) | Es un conjunto de símbolos. Para los lenguajes formales, utilizamos alfabetos que contienen una cantidad finita de símbolos. Ejemplos:<br>• $\sum_1$ = (a, b, c, d, e, f, g, h, i).<br>• $\sum_2$ = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).<br>• $\sum_3$ = El alfabeto del lenguaje de programación C. |
| Palabra | Es una secuencia finita de símbolos de un alfabeto ∑. Ejemplos:<br>• abc → es una palabra definida sobre $\sum_1$.<br>• 12 → es una palabra definida sobre $\sum_2$.<br>• main → es una palabra definida sobre $\sum_3$. |
| Palabra vacía (λ) | Es una palabra que no contiene símbolos (es una convención similar al número 0 en matemáticas). |


© Universidad Europea de Madrid. Todos los derechos reservados.

| | |
|---|---|
| Universo de un alfabeto W (∑) | Lo constituyen todas las palabras que pueden formarse con símbolos del alfabeto ∑, incluyendo a la palabra vacía (λ). Contiene, por tanto, un número infinito de palabras. Ejemplo: supongamos el alfabeto $\sum_1$ = (a, b, c, d, e , f, g, h, i). El universo de ese alfabeto sería: W ($\sum_1$) = {λ, a, aa, ba, ab, ac, ca, de,...}. |
| Lenguaje sobre un alfabeto L (∑) | Es cualquier subconjunto del universo de ese alfabeto, y por tanto, puede ser también infinito. Ejemplo: supongamos el alfabeto $\sum_1$ = (a, b, c, d, e , f, g, h, i). Posibles lenguajes de ese alfabeto podrían ser: <br>• $L_1\sum_1$ = {λ, aa, bb, cc, dd}. <br>• $L_2\sum_1$ ={ba, ca, da, fa, ga, ha}. |


© Universidad Europea de Madrid. Todos los derechos reservados.

## Operaciones con palabras

A partir de las palabras y de las operaciones que vamos a ver se pueden construir otras palabras.

Estas operaciones son: **concatenación, potencia y reflexión.**

| Concatenación | Potencia | Reflexión |
|---|---|---|
| Dadas dos palabras, α y β, de un mismo alfabeto ∑, la concatenación de estas dos palabras es una nueva palabra formada por los símbolos de α, seguido de los símbolos de β. Esta operación se representa mediante un punto α.β, que suele omitirse.<br><br>Ejemplo: sea ∑1 = (a, b, c, d, e , f, g, h, i), si α = ba y β = ca, = baca<br>• Esta operación no es conmutativa, puesto que β.α, da otra palabra diferente, en este caso sería, β.α = caba<br>• El elemento neutro de esta operación es la palabra vacía (λ). | Si concatenamos la misma palabra varias veces efectuamos a operación potencia. Esto sería la potencia i-ésima de esa palabra.<br><br>Ejemplo: sea $\sum_1$ = (a, b, c, d, e , f, g, h, i), si α = ba, $\alpha^2$ = α.α = baba.<br><br>• Por definición, cualquier palabra elevada a 0 es la palabra vacía, $\alpha^0$ = λ. | La palabra inversa de una dada se construye invirtiendo el orden de los símbolos de la palabra.<br><br>Ejemplo: sea $\sum_1$ = (a, b, c, d, e , f, g, h, i), si α = ba, $\alpha^{-1}$ = ab. |



© Universidad Europea de Madrid. Todos los derechos reservados.

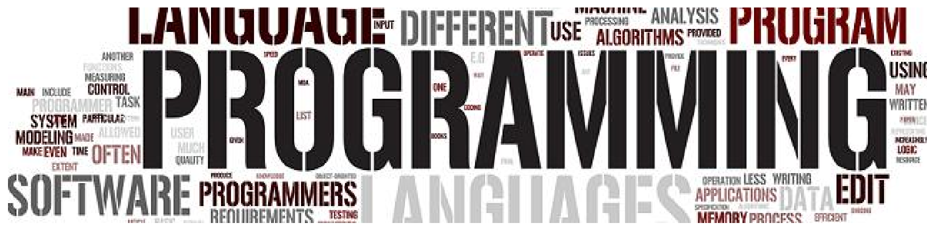
## Operaciones con lenguajes

Como hemos visto anteriormente un **lenguaje** es cualquier subconjunto del universo de un alfabeto L (∑), y las operaciones que nos interesan sobre un lenguaje son: **unión, concatenación,** clausura **positiva y la clausura o (cierre de Kleene).** Hay otras operaciones como **la potencia, reflexión, resta o complemento** que no las vamos a necesitar.

| | |
|---|---|
| Unión | Sean $L_1$ y $L_2$ dos lenguajes sobre un alfabeto ∑, $L_1$ U $L_2$ es el lenguaje formado por las palabras de $L_1$ y por las palabras de $L_2$.<br><br>Ejemplo: supongamos el alfabeto $\sum_1$ = (a, b, c, d, e, f, g, h, i) y dos lenguajes de ese alfabeto podrían ser: $L_1\sum_1$ = {λ, aa, bb, cc, dd} y $L_2\sum_1$ = {ba, ca, da, fa, ga, ha}. La unión de $L_1$ con $L_2$ sería:<br><br>• $L_1$ U $L_2$ = {λ, aa, bb, cc, dd, ba, ca, da, fa, ga, ha}, que cumple con la propiedad conmutativa, es decir que $L_2$ U $L_1$ daría el mismo resultado. |
| Concatenación | Sean $L_1$ y $L_2$ dos lenguajes sobre un alfabeto ∑, $L_1$. $L_2$ es el lenguaje formado por las palabras de $L_1$ concatenado con las palabras de $L_2$.<br><br>Ejemplo: a partir del alfabeto y los lenguajes descritos en el párrafo anterior, $L_1$. $L_2$ = {ba, ca, da, fa, ga, ha, aaba, aaca, aada,...}. |
| Cierre Positivo ($\sum^+$) | Es el lenguaje formado por todas las palabras que pueden formarse con símbolos del alfabeto ∑, excluyendo la palabra vacía. Si $L_2\sum_1$ = {ba, ca, da, fa, ga, ha}, el cierre positivo de ese lenguaje será $L_2$ $(\sum_1)^+$ = {ba, baca, bada, ca, caba, cada, cafa,...}. |


© Universidad Europea de Madrid. Todos los derechos reservados.

| Cierre o cerradura de Kleene (∑*) | Es el lenguaje formado por todas las palabras que pueden formarse con símbolos del alfabeto ∑, incluyendo la palabra vacía, que siempre será parte del cierre. Si $L_2 \Sigma_1 = \{ba, ca, da, fa, ga, ha\}$, el cierre de ese lenguaje será $L_2 (\Sigma_1)^*$ **=** {**λ**, ba, baca, bada, ca, caba, cada, cafa,...}. |
|---|---|




**Clausura**

A la operación de cierre también se le denomina clausura en distintos libros de texto.

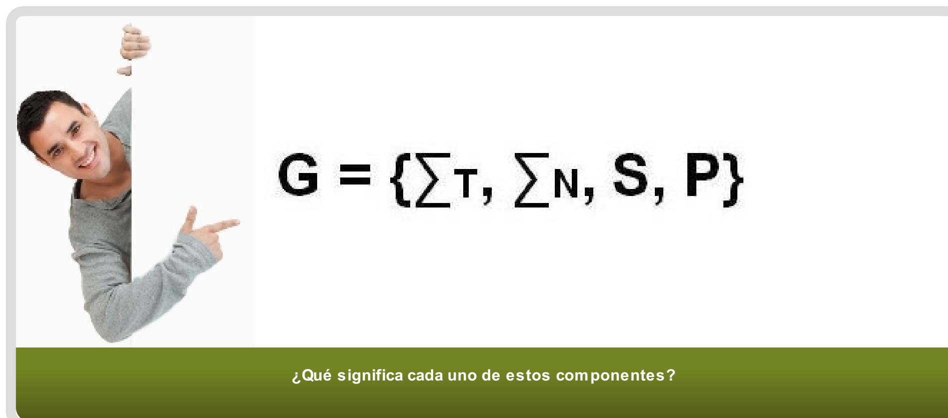


© Universidad Europea de Madrid. Todos los derechos reservados.

## Gramáticas formales

> Una gramática formal es una descripción de la estructura de las sentencias que forman un lenguaje, entendiendo por lenguaje, en este contexto, un lenguaje de programación.

La forma en la que se describe una gramática (G) es mediante cuatro términos: el alfabeto de los símbolos terminales ($\sum_T$), el alfabeto de los símbolos no terminales ($\sum_N$), el axioma o símbolo inicial de la gramática (S) y un conjunto de producciones (P). Se denota de la siguiente forma:




- **Símbolos terminales ($\sum_T$):** representa al conjunto de palabras reservadas de un lenguaje de programación. Esto incluye los caracteres especiales y los operadores tanto aritméticos como lógicos. Ejemplo: en el lenguaje de programación C: $\sum_T$ = {main, {, }, #, include, if, then, define, int, float, char, double;, +, -, *, >, >=, |, ...}. Se denotan con letras minúsculas: Ej: **int, float, if.**
- **Símbolos no terminales ($\sum_N$):** representa el conjunto de variables que utilizamos en las producciones de las gramáticas como símbolos de transición. Esto incluye el símbolo inicial de la gramática (S) y las variables que utilicemos. Se denota con letras mayúsculas y veremos un ejemplo cuando expliquemos las producciones.
- **Símbolo inicial (S):** es un símbolo no terminal a partir del cual se obtienen todas las palabras del lenguaje y será el primer símbolo de la gramática G. También se le denomina el **axioma** de la gramática.
- **Conjunto de producciones (P):** es un conjunto de reglas que indica cómo se obtienen las palabras del lenguaje definido por G. Establece las relaciones entre los símbolos terminales y los no terminales.

Reglas del conjunto de producciones (P)

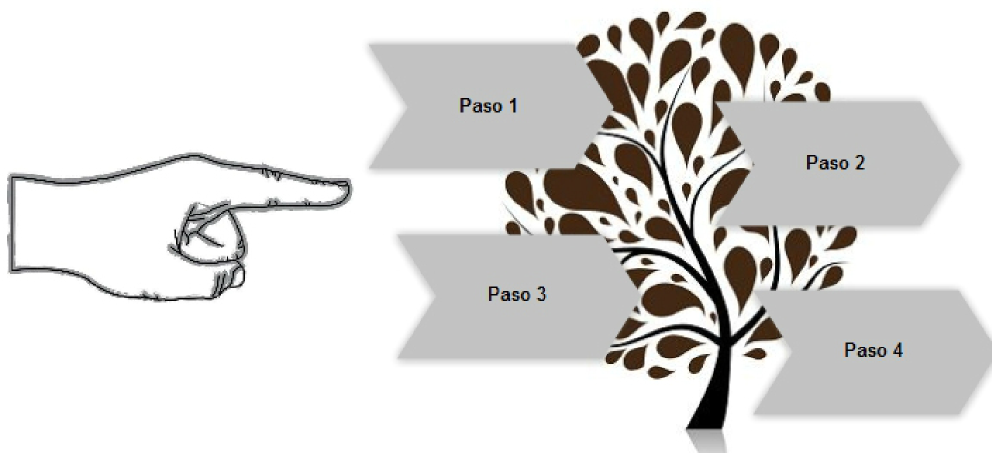

© Universidad Europea de Madrid. Todos los derechos reservados.

## Derivaciones y árbol de derivación

Hemos visto de qué forma se puede especificar una gramática para generar frases de un lenguaje específico, aunque un compilador no se utiliza para generar programas.

Un compilador debe revisar **los símbolos y las sentencias** que se construyen con ellos para determinar si pertenecen o no al lenguaje. Por tanto de lo que se trata es de ver cómo, a partir del símbolo inicial de la gramática se puede derivar una secuencia de símbolos utilizando las producciones de esa gramática, y esto se hace mediante **derivaciones.** Estas derivaciones se representan gráficamente mediante un árbol de derivación, también llamado **árbol de análisis sintáctico.**

Lo vamos a ir viendo con un ejemplo, para la gramática definida en (1) y con la frase (id * id):




**Paso 1**

El símbolo inicial de la gramática se representa como la raíz del árbol. En este caso E.

| Derivaciones | Árbol de derivación |
|---|---|
| 1.- E → (E) | E<br>/ \| \\<br>( E ) |



© Universidad Europea de Madrid. Todos los derechos reservados.

**Paso 2**

Derivamos E como nodo intermedio que será sustituido por E*E. Los nodos intermedios son símbolos no terminales de la gramática.

| Derivaciones | Árbol de derivación |
|---|---|
| 1.- E → E * E | E / \| \ ( E ) / \| \ E * E |

**Paso 3**

Derivamos otra vez E sustituyendo por el terminal id, obteniéndose los nodos hoja del árbol de derivación. Los nodos hoja son símbolos terminales de la gramática.

| Derivaciones | Árbol de derivación |
|---|---|
| 3.- E → id | E / \| \ ( E ) / \| \ E * E \| id |

**Paso 4**

Derivamos E del lado derecho, sustituyendo por el terminal id, obteniéndose el otro nodo hoja del árbol de derivación, dándose por finalizadas las derivaciones al reconocerse la frase inicial.

| Derivaciones | Árbol de derivación |
|---|---|
| 4.- E → id | E / \| \ ( E ) / \| \ E * E \| \| id id |



© Universidad Europea de Madrid. Todos los derechos reservados.

## Ambigüedad

La ambigüedad se produce cuando hay más de una manera de reconocer una palabra o sentencia a partir del símbolo inicial de la gramática.

En el ejemplo anterior (paso 3), hemos derivado E → id, empezando por la derivación más a la izquierda, pero también lo podríamos haber hecho derivando por el no terminal (E) que se encuentra más a la derecha, denominada derivación más a la derecha. En este caso, el resultado es el mismo, la frase ha sido reconocida, pero en otros casos dependiendo de la derivación que escojamos los resultados pueden variar. La ambigüedad se puede dar a varios niveles: **sentencia, gramática o lenguaje.**

| | |
|---|---|
| Sentencia | Una sentencia es ambigua si existe más de una derivación para reconocerla en una gramática. |
| Gramática | Una gramática es ambigua si existe en su lenguaje una sentencia ambigua, es decir, es posible derivar una sentencia por más de una derivación. |
| Lenguaje | Un lenguaje es ambiguo si existe una gramática ambigua que lo genera. Si todas las gramáticas que lo generan son ambiguas, entonces se denomina al lenguaje inherentemente ambiguo. |

El concepto es siempre el mismo, que a través de más de una derivación se puede obtener la sentencia, gramática o lenguaje a reconocer. Algunas veces esa ambigüedad se puede resolver, modificando la gramática directamente o analizando sus causas. Básicamente consiste en eliminar la recursividad "a izquierda o derecha" o factorizar por la izquierda.

| | |
|---|---|
| Recursividad por la izquierda | Se produce este tipo de recursividad cuando el primer símbolo no terminal aparece tanto en la parte derecha como en la izquierda de una producción. Ejemplo: **A** → **A**α, donde α representa cualquier combinación de terminales y no terminales (no tiene por qué ser un solo símbolo). Veremos más adelante como se elimina esta recursividad. |
| Recursividad por la derecha | Si el símbolo no terminal que aparece el último en la parte derecha de la producción es igual al de la parte izquierda. Ejemplo: A → αA |
| Factorizar por la izquierda | Esta solución se aplica cuando hay varias alternativas en una sentencia que comienzan igual. Ejemplo: A → Bα \| Bx, tenemos que eliminar esta ambigüedad reescribiendo las producciones. |


© Universidad Europea de Madrid. Todos los derechos reservados.

## Eliminar la ambigüedad

Ahora que conocemos las ambigüedades más típicas, vamos a aprender cómo eliminar la recursividad por la izquierda y las alternativas que comienzan igual, puesto que una gramática de este tipo es un problema para el analizador sintáctico.

### Eliminar la recursividad a izquierdas

Partimos de la siguiente producción $A \rightarrow A\alpha \mid \beta$, donde tanto $\alpha$ como $\beta$ representan conjuntos de terminales y no terminales. La regla a aplicar es la siguiente:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A \mid \lambda$$

Ejemplo: $E \rightarrow E + T \mid T$, donde $\alpha = + T$ y $\beta = T$. Por tanto, el resultado sería:

$$E \rightarrow T E'$$

$$E' \rightarrow + TE \mid \lambda$$

### Factorizar por la izquierda

Puesto que parte de alternativas de una producción que comienzan igual, tenemos que determinar la parte común más larga entre estas alternativas y a partir de ahí se sustituye por un no terminal que actuará de discriminante. Si tenemos $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid .... \mid \alpha\ \beta_n \mid \theta$, donde $\theta$ representa otras alternativas que no comienzan con $\alpha$, se hace lo siguiente:

$$A \rightarrow \alpha A' \mid \theta$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid .... \mid \beta_n$$



© Universidad Europea de Madrid. Todos los derechos reservados.

Ejemplo: $E \rightarrow S\ d\ E \mid S$ donde la parte común, α, es igual a S, por tanto aplicando la regla anterior:

$$E \rightarrow S\ A'$$

$$A' \rightarrow d\ E \mid \lambda$$



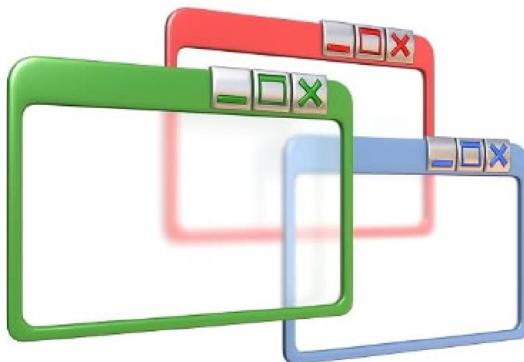
© Universidad Europea de Madrid. Todos los derechos reservados.

## Clasificación de los lenguajes de programación

> El objetivo de las gramáticas formales es definir los lenguajes de programación, y por tanto hay que conocer qué tipos de lenguajes hay y como se clasifican.

Por supuesto, hay varias formas de clasificar los miles de lenguajes que existen. Cueva Lovelle (1998) y otros autores los clasifican atendiendo a los siguientes criterios:

- Según su grado de independencia de la máquina.
- Según la forma de las instrucciones.
- Por generaciones.
- Según la forma de ejecución.



**Según su grado de independencia de la máquina**

**Según la forma de las instrucciones**

**Por generaciones**

**Según la forma de ejecución**

**Según el grado de independencia de la máquina**

- **Lenguaje máquina.**
- **Lenguaje ensamblador.**
- **Lenguajes de medio nivel:** ejemplo de este tipo es el lenguaje C puesto que tiene alguna de las características de los de bajo nivel junto con las que tienen los lenguajes de alto nivel.
- **Lenguajes de alto nivel:** aquí se agrupan la mayor parte de compiladores actuales, donde la abstracción de la máquina es total. Ejemplo de este tipo son: COBOL, Pascal, Java C++, C#, etc.
- **Lenguajes orientados a problemas concretos:** ejemplo de estos lenguajes son SQL, orientado a las bases de datos, SPARQL orientados a las consultas semánticas en grafos RDF, etc.

**Según la forma de las instrucciones**

- **Lenguajes imperativos o procedimentales:** son los que especifican como se va a realizar un cálculo, mientras que los declarativos especifican qué cálculo se va a realizar. Son lenguajes imperativos C, C++, C# y Java.
- **Lenguajes declarativos (lógicos y funcionales):** son lenguajes funcionales como ML, Haskell, junto con los de lógica de restricciones como Prolog.
- **Lenguajes concurrentes:** son los que permiten la ejecución simultánea de dos o más tareas. Ada es un ejemplo, junto con OCCAM o Concurrent Pascal entre otros.
- **Lenguajes orientados a objetos:** son los lenguajes que soportan la programación orientada a objetos (tipos abstractos de datos, clases, herencia y polimorfismo). Ejemplos típicos son Smaltalk, Eiffel, Ada 95, C++ y Java.


© Universidad Europea de Madrid. Todos los derechos reservados.

**Por generaciones**

- **Primera generación:** está constituida por los lenguajes máquina y ensamblador.
- **Segunda generación:** comienza con el desarrollo del compilador de FORTRAN y posteriormente con COBOL. Son lenguajes con asignación estática de memoria, sin recursividad ni estructuras dinámicas de datos.
- **Tercera generación:** están unidos a la programación estructurada. Son ejemplos Algol 60, Pascal, Modula y C.
- **Cuarta generación:** son lenguajes de muy alto nivel dedicados a tareas específicas. Son ejemplos SQL, DB2, JCL, etc.
- **Quinta generación:** está relacionada con los lenguajes orientados a la Inteligencia Artificial, como Lisp y Prolog.

**Según la forma de ejecución**

- **Compilados:** son lenguajes que están, como su propio nombre indica compilados, es decir se convierte a binario y se ejecutan tantas veces como sea necesario. La mayoría de los lenguajes actuales son compilados: C, C++, C#, Java, Pascal, etc. aunque también tienen su versión interpretada.
- **Interpretados:** cada vez que se usa el programa debe utilizarse un traductor denominado "intérprete" que se encarga de traducir las instrucciones del código fuente a código máquina según van siendo utilizadas. Ejemplos de estos lenguajes son BASIC, Python o Ruby.



© Universidad Europea de Madrid. Todos los derechos reservados.

## Ventajas e inconvenientes de los lenguajes de alto nivel

**Principales ventajas de los lenguajes de alto nivel con respecto a los de bajo nivel (Cueva Lovelle, 1998)**

- Son fáciles de aprender.
- No es necesario conocer cómo se gestionan los distintos tipos de datos por la memoria del ordenador.
- Gran número de estructuras de control (if-then-else, while, for, case, repeat, etc.).
- Se depuran más fácilmente.
- Tienen mayor capacidad de creación de estructuras de datos (estáticas y dinámicas).
- Permiten un diseño modular de los programas.

**Principales inconvenientes de los lenguajes de alto nivel**

- No se tiene acceso a ciertas zonas de la máquina.
- Reducción de la velocidad y de la optimización de los programas.
- El tamaño que ocupan y que aumenta según se van alejando de la máquina, para hacer lo mismo necesitan un gran número de bits.

Universidad Europea de Madrid

© Universidad Europea de Madrid. Todos los derechos reservados.

## Resumen

En este tema hemos dado un repaso a la historia de los lenguajes formales a partir del trabajo de Chomsky, Backus y Naur, que permitieron dar un salto hacia las gramáticas bien formadas o forma normal de Backus-Naur (BNF), lo que nos ha permitido ver su necesidad.

En primer lugar, hemos repasado las definiciones básicas: símbolo, alfabeto, palabra, palabra vacía, universo de un alfabeto y lenguaje desde le punto de vista de las gramáticas formales. Además de estas definiciones, hemos visto las operaciones más importantes que se pueden hacer con palabras (concatenación, potencia y reflexión) y con lenguajes (unión, concatenación, cierre positivo y cierre (o cierre de Kleene)).

Posteriormente, hemos aprendido a especificar un gramática utilizando sus cuatro componentes, $G = \{\Sigma_T, \Sigma_N, S, P\}$ y también hemos entendido cómo funcionan las derivaciones y los árboles de derivación (o árboles de análisis sintáctico) y con ello el problema de ambigüedad si la gramática no está bien especificada. En lo relativo a la ambigüedad, se ha visto cómo se elimina en el caso de una gramática recursiva por la izquierda, o con varias alternativas que comienzan por los mismos símbolos (factorización).

Para finalizar hemos dado un repaso a los distintos tipos de lenguajes atendiendo al grado de independencia de la máquina, la forma de las instrucciones, las generaciones o la forma de ejecución, además de entender las ventajas e inconvenientes de los lenguajes de alto nivel.



© Universidad Europea de Madrid. Todos los derechos reservados.