

MARZO 14, 2022

# ANÁLISIS SINTÁCTICO



# Análisis Sintáctico

Todo lenguaje de programación tiene reglas que describen la estructura sintáctica de programas bien formados.

En Pascal, por ejemplo, un programa se compone de bloques, un bloque de proposiciones, una proposición de expresiones, una expresión de componentes léxicos, y así sucesivamente.

Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de **gramáticas de contexto libre** o notación **BNF** ( Backus-Naur Form).

# Análisis Sintáctico

Las gramáticas ofrecen ventajas significativas a los diseñadores de lenguajes y a los desarrolladores de compiladores.

- ✓ Las gramáticas son especificaciones sintácticas y precisas de lenguajes de programación.
- ✓ A partir de una gramática se puede generar automáticamente un analizador sintáctico.
- ✓ El proceso de construcción puede llevar a descubrir ambigüedades.
- ✓ Una gramática proporciona una estructura a un lenguaje de programación, siendo más fácil generar código y detectar errores.
- ✓ Es más fácil ampliar/modificar el lenguaje si está descrito con una gramática.

# El papel del analizador sintáctico

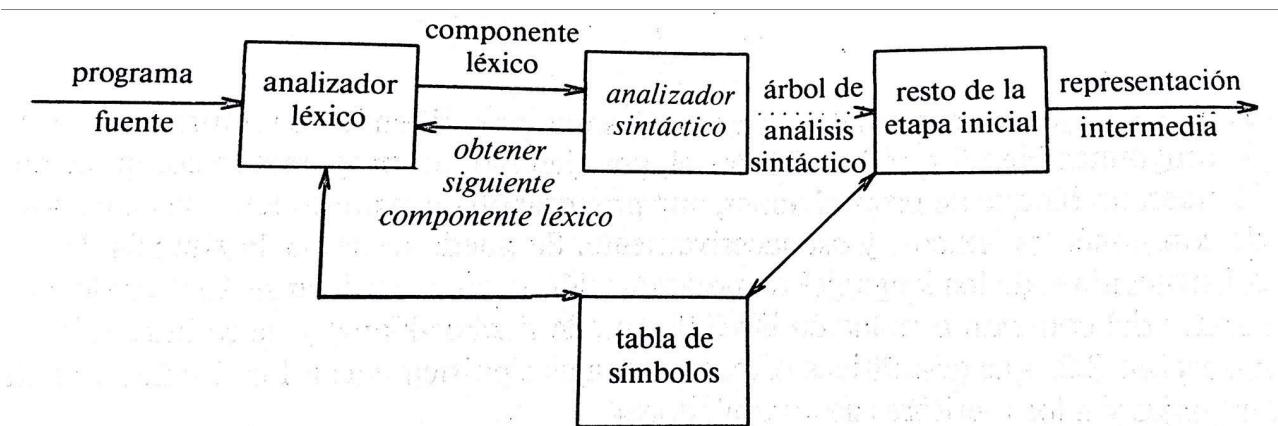
El analizador sintáctico obtiene una cadena de componentes léxicos del analizador léxico y comprueba si la cadena puede ser generada por la gramática del lenguaje fuente.

El analizador sintáctico informará de cualquier error de sintaxis de manera inteligible.

También debería recuperarse de los errores que ocurren con frecuencia para poder continuar procesando el resto de la entrada.

We expect the parser to recover from commonly occurring errors to continue processing the remainder of the program.

# Posición del analizador sintáctico en el modelo del compilador



## ¿Qué es el analizador sintáctico ?

Es la fase del analizador que se encarga de revisar el texto de entrada en base a una gramática dada. Y en caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce.

En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de tokens suministrada por el analizador léxico.

## El analizador sintáctico también hace:

- ❑ Acceder a la tabla de símbolos (para hacer parte del trabajo del analizador semántico).
- ❑ Chequeo de tipos ( del analizador semántico).
- ❑ Generar código intermedio.
- ❑ Generar errores cuando se producen.

En definitiva, realiza casi todas las operaciones de la compilación. Este método de trabajo da lugar a los métodos de compilación dirigidos por sintaxis.

## Tipos generales de analizadores sintácticos para gramáticas

: three types : 1) universal

2) top - down

3) bottom - up

Los algoritmos universales de análisis sintático, como el algoritmo de Cocke-Younger-Kasami y el de Early, pueden analizar cualquier gramática, pero ineficientes para utilizarlos en la producción de compiladores.

Los métodos empleados generalmente en los compiladores se clasifican como **descendentes** o **ascendentes**.

↳ top - down

↳ bottom up

# Tipos generales de analizadores sintácticos para gramáticas

**Top - down methods :**

Los analizadores sintácticos **descendentes** construyen árboles de análisis sintáctico desde la raíz hasta las hojas.

**Bottom - up methods :**

Los analizadores sintácticos **ascendentes** construyen árboles de análisis sintáctico desde las hojas y suben hacia la raíz.

En ambos casos se examina la entrada al analizador sintáctico de izquierda a derecha, un símbolo a la vez.

## Métodos descendentes y ascendentes

Los métodos descendentes y ascendentes más eficientes trabajan <sup>only for</sup> ~~coh~~ subclases de gramáticas, pero varias de éstas subclases, como gramáticas LL y LR, son lo suficientemente expresivas para describir la mayoría de las construcciones sintácticas de los lenguajes de programación.

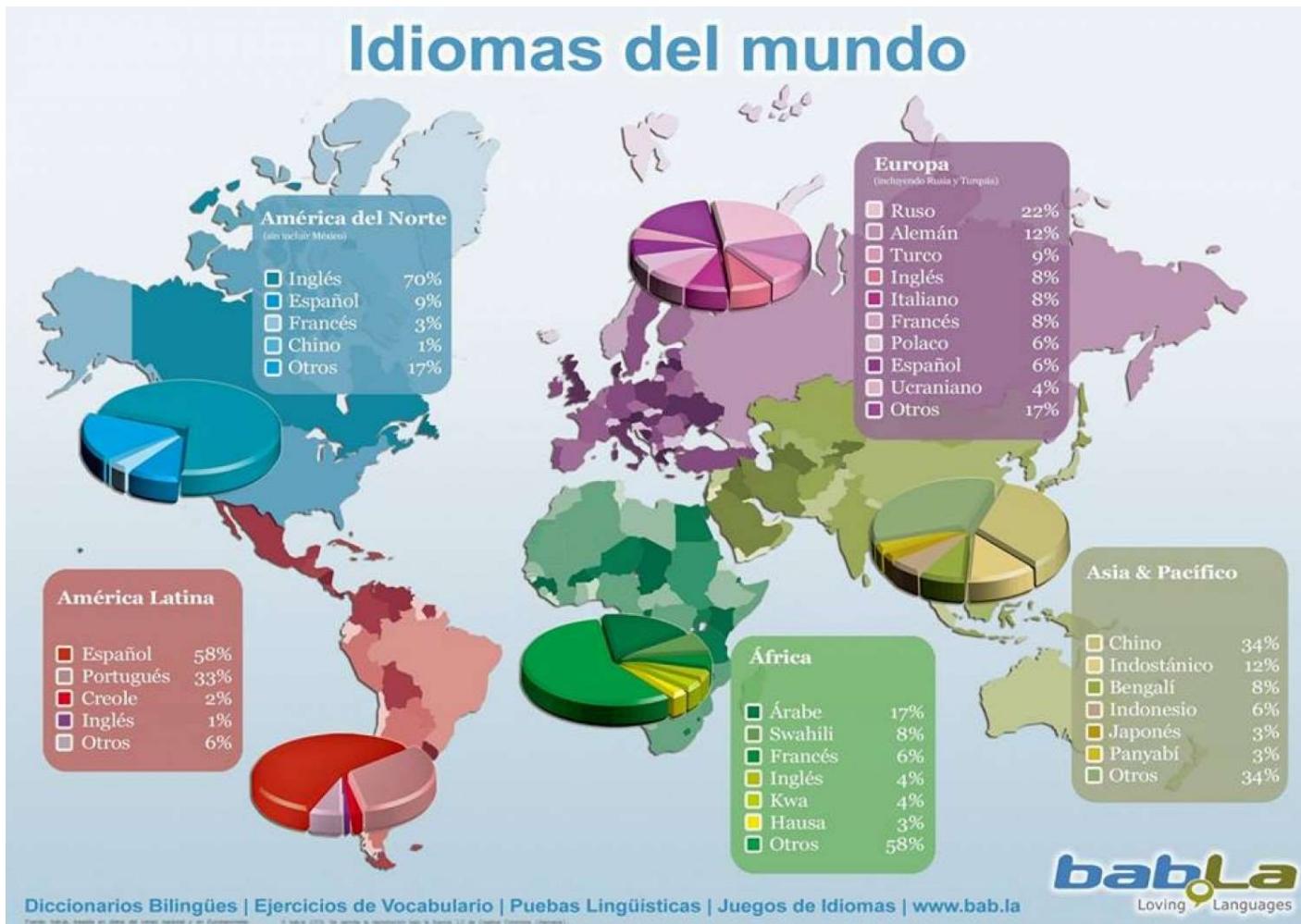
## Actividad grupal

Elaborar una presentación de máximo 6 diapositivas (incluida la portada y las fuentes), donde explique lo que es:

una gramática LL (Equipo 1) y  
una gramática LR (Equipo 2).



# Idiomas del mundo



# TIPOS DE ERRORES

Universidad Panamericana

# Tipos de errores

Los errores en la programación pueden ser de los siguientes tipos:

→ misspellings

- ✗ Léxicos, producidos al escribir mal un identificador, una palabra clave o un operador. i.e. elipsesize instead of ellipseSize, missing quotes, etc.
- ✗ Sintácticos, por una expresión aritmética o paréntesis no equilibrados. Misplaced semicolons or extra/missing braces ({} )
- ✗ Semánticos, como un operador aplicado a un operando incompatible.
- ✗ Lógicos, como una llamada infinitamente recursiva.

→ type mismatches between operators and operands . I.e. having a return value in a void func

Universidad Panamericana  
Can be anything from incorrect reasoning from the programmer, use = instead of == for comparison. The program containing = may be well formed, but it does not reflect the programmer's intent.

when the stream of tokens

Parsing methods: LL and LR, detect errors as soon as possible : cannot be parsed further

Syntax error recovery strategies:

- 1) Panic mode
- 2) Phase-level recovery

## Manejo de errores sintácticos

Nos interesa que cuando el compilador encuentre un error, se recupere y siga buscando errores. Por lo tanto el manejador de errores de un analizador sintáctico debe tener como objetivos:

✓ Indicar los errores de forma clara y precisa. Aclarar el tipo de error y su localización.

✓ Recuperarse del error, para poder seguir examinando la entrada.

✓ No ralentizar significativamente la compilación.

→ Recover from each error quickly enough to detect subsequent errors.

→ Add minimal overhead to the processing of correct programs.

How should an error handler report an error? At least report the place in the source where it was detected: there's a chance the error occurred within the previous tokens.

#### 4.1.4 Error Recovery Strategies

Once an error is detected, how should the parser recover?

- ↳ The simplest approach is for the parser to quit with an informative error message when it detects the 1<sup>st</sup> error.
- ↳ The parser restores itself to a state where processing is correct again. → input char
- ↳ Strategies: panic-mode, phrase-level, error-productions and global-correction.

## Manejo de errores sintácticos

Un buen compilador debe hacerse siempre teniendo también en mente los errores que se pueden producir; con ello se consigue:

- ✓ Simplificar la estructura del compilador.
- ✓ Mejorar la respuesta ante los errores.

## Manejo de errores sintácticos

En algunos casos, un error pudo haber ocurrido mucho antes de la posición en que se detectó su presencia, y puede ser muy difícil deducir la naturaleza precisa del error.

En los casos difíciles, el manejador de errores quizá tenga que adivinar qué tenía en mente el programador cuando escribió el programa.

Los métodos LL y LR detectan un error lo antes posible, tienen la propiedad del **prefijo viable**.

LL and LR are parsing methods that detect errors as soon as possible. They have the viable-prefix property

## Manejo de errores sintácticos

### Prefijo viable

Detectan la presencia de un error nada más de ver un prefijo de la entrada que no es prefijo de ninguna cadena del lenguaje.



they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

... mas estrategias para corrección de errores.

### ***Ignorar el problema (Panic mode )***

Consiste en ignorar el resto de la entrada hasta llegar a una condición de seguridad.

Una condición tal se produce cuando nos encontramos un token especial (por ejemplo un ';' o un 'END').

A partir de este punto se sigue analizando normalmente.

## Ejemplo:

`aux = a[i]`

$a[i] = a[i];$

a[j] = aux;

`id '=' id '[' id ']' id '[' id ']' '=' id '[' id ']' ';' id '[' id ']' '=' id ';'`

## Token especial



... mas estrategias para corrección de errores.

### *Recuperación a nivel de frase*

Intenta recuperar el error una vez descubierto. En el caso anterior, por ejemplo, podría haber sido lo suficientemente inteligente como para insertar el token ‘;’ . Hay que tener cuidado con este método, pues puede dar lugar a recuperaciones infinitas.

## ... mas estrategias para corrección de errores.

### *Reglas de producción adicionales para el control de errores*

La gramática se puede aumentar con las reglas que reconocen los errores más comunes. En el caso anterior, se podría haber puesto algo como:

```
sent_errónea -> sent_sin_acabar sentencia_acabada ';  
sentencia_acabada -> sentencia ';'  
sent_sin_acabar -> sentencia
```

Lo cual nos da mayor control en ciertas circunstancias.

## ... mas estrategias para corrección de errores.

### *Corrección Global*

Dada una secuencia completa de tokens a ser reconocida, si hay algún error por el que no se puede reconocer, consiste en encontrar la secuencia completa más parecida que sí se pueda reconocer.

Es decir, el analizador sintáctico le pide toda la secuencia de tokens al léxico, y lo que hace es devolver lo más parecido a la cadena de entrada pero sin errores, así como el árbol que lo reconoce.

# GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO

<https://ccc.inaoep.mx/~emorales/Cursos/Automatas/GramsLibresContexto.pdf>

## GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO

Si  $S_1$  y  $S_2$  son proposiciones y E es una expresión, entonces

“ if E then  $S_1$  else  $S_2$ ” es una proposición

# GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO

La gramática con las siguientes producciones define expresiones aritméticas simples.

*expr -> expr op expr*

*expr -> (expr)*

*expr -> - expr*

*expr -> id*

*op -> +*

*op -> -*

*op -> \**

*op -> /*

*op -> ^*

Símbolos terminales?  
Símbolos no terminales?  
Símbolo inicial?  
Qué es una producción?

# Convenciones de notación

to avoid "these are terminals", we have conventions

## 1. Símbolos terminales

1. Las primeras letras del alfabeto minúsculas a, b, c,...
2. Los símbolos de operadores
3. Los símbolos de puntuación como paréntesis, coma, etc.
4. Dígitos
5. Cadenas en negritas como **if, si, no, ...**

## 2. Símbolos no terminales

1. Las primeras letras del alfabeto mayúsculas A, B , C...
2. Letra S, usualmente, símbolo inicial.
3. Nombres en cursivas minúsculas

## Convenciones de notación

3. Últimas letras mayúsculas del alfabeto , X, Y, Z, *símbolos gramaticales*, terminales y no terminales.  
*that is*
4. Últimas letras minúsculas del alfabeto, u, v, w, x, y, z, representan cadenas terminales. (*possibly empty*)
5. Las letras griegas minúsculas,  $\alpha, \beta, \gamma$ , cadenas de símbolos gramaticales. Entonces una producción genérica podría escribirse como  $A \rightarrow \alpha$  indicando que hay un solo no terminal A a la izquierda de la producción y una cadena de símbolos gramaticales  $\alpha$  a la derecha de la producción. *A is the head and  $\alpha$  the body*

## Convenciones de notación

with a common head ( $A$ ) : can be grouped with " | " (or)

6. Si  $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_n$ , son todas producciones con  $A$  a la izquierda y se puede escribir como alternativas de  $A$ ,  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ .
7. A menos que se diga otra cosa, el símbolo inicial de la primera producción es el símbolo inicial.

↳ the head of the first production is the start symbol

The construction of a parse tree can be made precise by taking a derivational view, where productions are treated as rewriting rules.

- ↳ Beginning with start symbol, each rewriting step replaces a nonterm by the body of one of its productions: this deriv view corresponds to the top-down construction of a tree.
- Bottom-up parsing: rightmost derivations, where the rightmost nonterminal is rewritten each step.
- Top-down parsing: leftmost derivation, where leftmost nonterminal is rewritten each step.

## Derivaciones

La idea central es que se considera una producción como una regla de reescritura, donde el no terminal de la izquierda es sustituido por la cadena del lado derecho de la producción.

Derivación por la izquierda : Derivación donde solo el no terminal de más a la izquierda de cualquier forma de frase se sustituye en cada paso.

Derivación por la derecha o Derivación canónica: Derivación donde el no terminal más a la derecha se sustituye en cada paso.

Example: consider the following grammar (one nonterminal, E) :

$$E \rightarrow E+E \mid E * E \mid -E \mid (E) \mid id \quad (4)$$

The production  $E \rightarrow -E$  says that  $E$  is an expression and  $-E$  as well. Rewriting  $-E$  as  $E$  is like saying " $E$  derives  $-E$ ". We can take a single  $E$  and repeatedly apply productions in any order to get a sequence of replacements:  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

$\xrightarrow{*}$ : derives in  $0^+$  steps  
 $\xrightarrow{+}$ : derives in  $1^+$  steps

"derivation of  $-(id)$  from  $E$ ": proof that  $-(id)$  is an instance of an expression

## Ejemplo: Sea la gramática

$E \rightarrow E + E$   
|  $E - E$   
|  $E * E$   
|  $E / E$   
|  $E ^ E$   
| id

Constrúyase una derivación por la izquierda y por la derecha para la siguiente frase:

$$a * c + b \equiv (id_1 * id_2 + id_3)$$

Derivación más a la izquierda:

$$E \rightarrow E + E \rightarrow E * E + E \rightarrow id_1 * E + E \rightarrow id_1 * id_2 + E \rightarrow id_1 * id_2 + id_3$$

Derivación más a la derecha:

$$E \rightarrow E + E \rightarrow E + id_3 \rightarrow E * E + id_3 \rightarrow E * id_2 + id_3 \rightarrow id_1 * id_2 + id_3$$

- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.
- Each interior node represents the application of a production
- The leaves are terminals, and from Left to Right, constitute the yield/frontier of the tree.

## Ambigüedad

- ✓ Una gramática que produce más de un árbol de análisis sintáctico es ambigua.
- ✓ Una gramática ambigua es la que produce más de una derivación por la izquierda o por la derecha para la misma frase.

# Prioridad

- ( )
- - (unitario)
- ^
- \*, /, %
- +, -
- <, >, <=, >=, <>, =
- no
- y
- o

