

INSTITUTO  
TECNOLOGICO  
DE  
CELAYA

Lenguajes y autómatas  
Equipo 2

Gomez Cabañas Anthony	20030057
Ortega Hernandez Cristhian Alberto	20031091
Ponce Morales Alma Gabriela	20030274
Ruiz Lopez Miguel Angel	20030935

Actividad

4



## DEPARTAMENTO DE SISTEMAS COMPUTACIONALES E INFORMÁTICA

ASUNTO: **SOLICITUD DE ACTIVIDADES**

Celaya, Guanajuato, 18 / septiembre / 202

### LENGUAJES Y AUTÓMATAS II

DOCENTE DESIGNADO: ISC. RICARDO GONZÁLEZ GONZÁLEZ  
**SEMESTRE AGOSTO-DICIEMBRE 2023**

**ACTIVIDAD 4 (VALOR 44 PUNTOS)**

LEA CUIDADOSAMENTE, Y REALICE LAS SIGUIENTE ACTIVIDADES, CONSIDERANDO LOS CRITERIOS DE CALIDAD PROPUESTOS EN LOS DOCUMENTOS DE LA [GUÍA TUTORIAL](#), Y LA [RÚBRICA DE EVALUACIÓN](#),

EL LECTOR DEBE TOMAR MUY EN CUENTA QUE ESTA ACTIVIDAD ES UN EXAMEN, Y NO UNA SIMPLE TAREA, PUES DEMANDA DEDICACIÓN PARA INVESTIGAR, LEER, ANALIZAR, REDACTAR, ILUSTRAR Y PROPOSER DE MANERA PROFESIONAL LOS TEMAS PROPUESTOS EN LA ESTRUCTURA TEMÁTICA DE ESTA ASIGNATURA.

#### 2. ANÁLISIS LÉXICO.

INVESTIGUE, LEA, COMPREnda Y ELABORE UNA **MONOGRAFÍA TÉCNICA** COMPLETAMENTE APEGADA A LO SOLICITADO EN LA [GUÍA TUTORIAL](#) (PUNTO 3, INCISO a ) ACERCA DE LOS SIGUIENTES TEMAS :

- GENERADORES DE ANALIZADORES LÉXICOS
- APPLICACIONES ( CASO DE ESTUDIO )

CONSIDERACIÓN :

DEBE USTED ENTENDER EL VALOR QUE TIENE ESTA ACTIVIDAD Y QUE LOS TEMAS ANTES REFERIDOS, PARA NADA DEBEN SER ABORDADOS COMO SIMPLES CONCEPTOS REDACTADOS CON LA LIGEREZA QUE YA SE HA OBSERVADO EN ACTIVIDADES PREVIAS.

ANALICE CADA TEMA, SUS CARACTERÍSTICAS, SU IMPORTANCIA, SUS CONCEPTOS, SUS EJEMPLOS, SUS ILUSTRACIONES, Y LOS TIPOS DE EVIDENCIAS QUE USARÁ PARA DEMOSTRAR QUE USTED HA ADQUIRIDO UN VERDADERO CONOCIMIENTO ACERCA DE ÉSTOS.



Av. Antonio García Cubas #600 esq. Av. Tecnológico, Colonia Alfredo V. Bonfil, C.P. 38010  
Celaya, Gto. Tel. 01 (461) 611 75 75 e-mail: lince@celaya.tecnm.mx tecnm.mx | celaya.tecnm.mx





**IMPORTANTE :** SI LO REQUIERE PUEDE CONSULTAR EL [SIGUIENTE DOCUMENTO](#) PARA ORIENTAR SU TRABAJO EN CONOCER QUÉ ES Y CÓMO HACER UNA MONOGRAFÍA CON EL RIGOR ACADÉMICO REQUERIDO.

EN LO REFERENTE AL TEMA **APLICACIONES ( CASO DE ESTUDIO )**, ARRIBA REFERIDO Y A MANERA DE PRÁCTICA, ELABORE CON EL APOYO DEL SOFTWARE LIBRE DENOMINADO **ANALIZADOR LÉXICO FLEX**, UN EJERCICIO DEMOSTRATIVO SOBRE AL MENOS TRES DEFINICIONES LÉXICAS PROPIAS DEL LENGUAJE PROGRAMACIÓN C.

### 3. ANÁLISIS SINTÁCTICO.

INVESTIGUE, LEA, COMPREnda Y ELABORE UNA **MONOGRAFÍA TÉCNICA** COMPLETAMENTE APEGADA A LO SOLICITADO EN LA [GUÍA TUTORIAL](#) (PUNTO 3, INCISO a ) ACERCA DE LOS SIGUIENTES TEMAS :

- GRAMÁTICAS LIBRES DE CONTEXTO Y ÁRBOLES DE DERIVACIÓN
- DIAGRAMAS DE SINTAXIS

CONSIDERACIÓN :

DEBE USTED ENTENDER EL VALOR QUE TIENE ESTA ACTIVIDAD Y QUE LOS TEMAS ANTES REFERIDOS, PARA NADA DEBEN SER ABORDADOS COMO SIMPLES CONCEPTOS REDACTADOS CON LA LIGEREZA QUE YA SE HA OBSERVADO EN ACTIVIDADES PREVIAS.

ANALICE CADA TEMA, SUS CARACTERÍSTICAS, SU IMPORTANCIA, SUS CONCEPTOS, SUS EJEMPLOS, SUS ILUSTRACIONES, Y LOS TIPOS DE EVIDENCIAS QUE USARÁ PARA DEMOSTRAR QUE USTED HA ADQUIRIDO UN VERDADERO CONOCIMIENTO ACERCA DE ÉSTOS.

A MODO DE PRÁCTICA REALICE ESTE PUNTO Y ELABORE EJERCICIOS PRÁCTICOS CON LOS CUÁLES USTED DEMUESTRE

- ¿ QUÉ SON LAS GRAMÁTICAS LIBRES DE CONTEXTO ?
- ¿ QUÉ ES UN CONTEXTO ?, HAGA UNA MUY BUENA ILUSTRACIÓN DE ESTE CONCEPTO.
- ¿ SIRVE DICHO CONTEXTO A LOS PROPÓSITOS DE UNA GRAMÁTICA QUE DEFINA UN LENGUAJE FORMAL ?

ADEMÁS INCLUYA EJEMPLOS ( AL MENOS TRES ) DE ÁRBOLES DE DERIVACIÓN Y DEMUESTRE CÓMO AYUDAN A LOS PROCESOS DE ANÁLISIS SINTÁCTICOS.

**IMPORTANTE :** SI LO REQUIERE PUEDE CONSULTAR EL [SIGUIENTE DOCUMENTO](#) PARA ORIENTAR SU TRABAJO EN CONOCER QUÉ ES Y CÓMO HACER UNA MONOGRAFÍA CON EL RIGOR ACADÉMICO REQUERIDO.

POR ÚLTIMO, RECUERDE LEER LA [GUÍA TUTORIAL](#) PARA EL CORRECTO TRATAMIENTO DE ESTE INCISO.



## ¿QUÉ SE CALIFICARÁ ?

LA RÚBRICA PARA EVALUAR ESTA ACTIVIDAD ESTARÁ INTEGRADA POR LOS SIGUIENTES CRITERIOS.

- a. **LA OPORTUNIDAD.** SI EL TRABAJO FUE ENTREGADO OPORTUNAMENTE.
- b. **LA COMPRENSIÓN.** SE VALORARÁ EL GRADO DE COMPRENSIÓN DEL TEMAS ANALIZADOS.
- c. **LA CALIDAD.** SI LAS EVIDENCIAS ENVIADAS CORRESPONDEN A LA CALIDAD ESPERADA PARA ESTE NIVEL PROFESIONAL QUE SE CURSA.
- d. **LA CAPACIDAD DE SÍNTESIS.** SI LAS EVIDENCIAS ENTREGADAS TIENEN EL NIVEL DE DETALLE Y PROFUNDIDAD REQUERIDA, O EN BIEN SI SE OMITIERON CONCEPTOS CON EL AFÁN DE SIMPLIFICAR Y ENTREGAR UN MATERIAL ACADÉMICA Y TÉCNICAMENTE POBRE.
- e. **LA CREATIVIDAD.** LA MANERA EN QUE SE EXPRESAN LOS CONCEPTOS Y EL TRATAMIENTO QUE SE DA A LA INFORMACIÓN ANALIZADA PARA QUE ÉSTA SEA COMPRESIBLE EN SU ESENCIA.

**IMPORTANTE :** CUENTA CON EL TIEMPO SUFFICIENTE PARA REALIZAR ESTA ACTIVIDAD Y SUMAR PUNTOS IMPORTANTES A SU CALIFICACIÓN DE ESTA EVALUACIÓN.

**IMPORTANTE :** TODO EL MATERIAL ESCRITO DEBERÁ SER HECHO A MANO.





## **CONSIDERACIONES.**

CADA UNO DE LOS PUNTOS ANTERIORES DEBE SER DESARROLLADO CON LA PROFUNDIDAD ACORDE A UN NIVEL PROFESIONAL, Y APEGÁNDOSE COMPLETAMENTE A LAS DIRECTRICES DE LA GUÍA TUTORIAL.

NO CONCIBA ESTE TRABAJO, COMO UN SIMPLE RESUMEN O EJERCICIO DE TRANSCRIPCIÓN, PUES EL VALOR INDICADO AL INICIO DE ESTA ACTIVIDAD LE DARÁ A USTED UNA BUENA IDEA DE LO QUE SE ESPERA DE ELLA, EN CUANTO A CALIDAD Y EL APRENDIZAJE OBTENIDO, MISMO QUE SERÁ PUESTO A PRUEBA MEDIANTE UN EXAMEN ESCRITO O BIEN ORAL EN CLASE.

SI DECIDIÓ ELABORAR ESTA ACTIVIDAD EN EQUIPO, CADA INTEGRANTE DE ÉSTE DEBERÁ POSEER EL MISMO NIVEL DE CONOCIMIENTO, PUES TAN SOLO REPARTIR TEMAS ENTRE LOS INTEGRANTES DEL EQUIPO, SUPONDRÍA UN GRAVE ERROR DE INTERPRETACIÓN A LA INTENCIÓN DIDÁCTICA REAL DE ESTA ACTIVIDAD.

POR ÚLTIMO, ESTA ACTIVIDAD SOLO SE PODRÁ DESARROLLAR EN EQUIPO, SI SE REGISTRÓ EN UNO PREVIAMENTE, UTILIZANDO EL FORMATO ENTREGADO EN LA ACTIVIDAD INICIAL. DE LO CONTRARIO DEBERÁ ELABORAR Y ENTREGAR LA ACTIVIDAD DE FORMA INDIVIDUAL.

LA ENTREGA DE DICHO REGISTRO SE HARÁ VÍA CORREO ELECTRÓNICO ENVIANDO ÉSTE AL PROFESOR DESIGNADO, Y POSTERIORMENTE EN CLASE ENTREGANDO LA HOJA EN FÍSICO.

## **OBSERVACIONES:**

- CADA HOJA QUE ENTREGUE DE SU ACTIVIDAD, DEBERÁ ESTAR FIRMADA AL MARGEN DERECHO, INCLUIDA LA PROPIA SOLICITUD DE LA ACTIVIDAD.
- INTEGRE TODO SU TRABAJO EN UN SOLO ARCHIVO DE TIPO .PDF, Y ASIGNE EL NOMBRE QUE A CONTINUACIÓN SE INDICA.

NO OLVIDE ANEXAR LAS HOJAS DE ESTA ACTIVIDAD Y DE SU TRABAJO DESPUÉS DE SU PORTADA.

- UNA VEZ ELABORADA SU ACTIVIDAD, RECUERDE DIGITALIZARLA Y NOMBRARLA EN BASE A LA NOMENCLATURA QUE SE INDICA MÁS ADELANTE EN ESTE DOCUMENTO.
- SI SUS EVIDENCIAS ENVIADAS POR CORREO, NO CUMPLEN CON LA NOMENCLATURA SOLICITADA, NO SERÁN CONSIDERADAS COMO EVIDENCIAS PARA SU EVALUACIÓN.
- POR ÚLTIMO, POR FAVOR GESTIONE APROPIADAMENTE SU TIEMPO, Y SEA PUNTUAL EN SU ENTREGA Y ASÍ EVITAR PROBLEMAS DE NULIDAD POR EXTEMPORANEIDAD.





**LA NOMENCLATURA SOLICITADA PARA ENVIAR SU TRABAJO ES LA SIGUIENTE :**

AAAA-MM-DD\_TNM\_CELAYA\_MATERIA\_DOCUMENTO\_[EQUIPO]\_NOCTROL\_APELLIDOS\_NOMBRE\_SEM.PDF

**( NOTA : \*\*\* TODO DEBE SER ESCRITO USANDO LETRAS MAYÚSCULAS \*\*\* )**

**DONDE :**

TNM_CELAYA	: INSTITUCIÓN ACADÉMICA
AAAA	: AÑO
MM	: MES
DD	: DÍA
MATERIA	: <b>LAI</b> , LI MÁS EL GRUPO ( -A, -B, -C )
DOCUMENTO	: AI-ACTIVIDAD 1, P1-PRACTICA 1, R1-REPORTE 1, T1-TAREA 1, PG1-PROGRAMA, ETC. (CAMBIANDO EL NÚMERO CONSECUТИVO POR EL QUE CORRESPONDA)
[EQUIPO]	: NÚMERO DEL EQUIPO QUE CORRESPONDA SEGÚN INDICACIÓN DEL PROFESOR. [OPCIONAL]
NOCTROL	: SU NÚMERO DE CONTROL
APELLIDOS	: SUS APELLIDOS
NOMBRE	: SU NOMBRE
SEM	: EL PERÍODO SEMESTRAL EN CURSO: <b>AGO-DIC</b>

**EJEMPLO :**

SI EL TRABAJO SE SOLICITÓ EN EQUIPO.

2023-09-18\_TNM\_CELAYA\_LAI-A\_A4\_EQUIPO\_99\_9999999\_PEREZ\_PEREZ\_JUAN\_AGO-DIC23.PDF

DONDE EL NOMBRE DEBERÁ CORRESPONDER AL JEFE DE EQUIPO QUE HACE LA ENTREGA DEL TRABAJO.

SI EL TRABAJO SE SOLICITÓ INDIVIDUALMENTE.

2023-09-18\_TNM\_CELAYA\_LAI-A\_A4\_9999999\_PEREZ\_PEREZ\_JUAN\_AGO-DIC23.PDF





**FECHA Y HORA DE ENTREGA:**

LA INDICADA EN LA PLATAFORMA VIRTUAL.

EN CASO DE QUE EL TRABAJO SE HAYA SOLICITADO EN EQUIPO, EL JEFE DEL MISMO SERÁ EL ÚNICO RESPONSABLE DE ENVIAR LA ACTIVIDAD EN LA PLATAFORMA VIRTUAL.

**MUY IMPORTANTE:**

1. DESPUÉS DE LA HORA INDICADA EN LA PLATAFORMA VIRTUAL ( AÚN CUANDO SOLO SEA UN MINUTO O VARIOS ), LA ACTIVIDAD SERÁ CONSIDERADA COMO EXTEMPORÁNEA Y NO CONTARÁ COMO EVIDENCIA PARA SU EVALUACIÓN.

SE LE SUGIERE ENVIAR CON ANTICIPACIÓN SU ACTIVIDAD A FIN DE EVITAR CONFLICTOS POR NO ENTREGAR ÉSTA A TIEMPO.

BAJO NINGÚN PRETEXTO O JUSTIFICACIÓN SE ACEPTARÁN LOS TRABAJOS EXTEMPORÁNEOS, EVITE LA PENA DE RECORDAR A USTED QUE EL VALOR DE LA PUNTUALIDAD ES PARTE IMPORTANTE DE SUS EVIDENCIAS Y ES EL PRIMER PUNTO QUE SE HA DE EVALUAR.

2. NO OLVIDE ANEXAR A SU ARCHIVO .PDF DE EVIDENCIAS UNA PORTADA PROFESIONAL, Y ESTA SOLICITUD DE ACTIVIDADES CON TODAS LAS HOJAS FIRMADAS EN EL MARGEN DERECHO.
3. POR ÚLTIMO, TODA EVIDENCIA GENERADA QUE CONTENGA AL MENOS UNA TRANSCRIPCIÓN DE CUALQUIER FUENTE Y DE CUALQUIER TIPO, ES DECIR CON MATERIAL PLAGIADO SERÁ ANULADA DE FORMA INCONTROVERTIBLE.



*[Handwritten signatures]*

# INSTITUTO TECNOLOGICO DE CELAYA

Lenguajes y Autómatas II

Equipo 2

Gomez Cabañas Anthony 20030057

Ortega Hernandez Cristhian Alberto 20031091

Ponce Morales Alma Gabriela 20030274

Ruiz Lopez Miguel Angel 20030935

## Monografía 1:

# Generadores de Analizadores Léxicos

I.S.C. Ricardo González González

# ÍNDICE

A. M. C.

## Índice de figuras

### Introducción

### Generalidades

#### 1. Compiladores

##### 1.1. Estructura de un compilador

##### 1.1.1. Fase de un compilador

#### 2. Áncilador léxico

##### 2.1. Expresiones regulares

##### 2.1.1. Operadores de expresiones regulares

##### 2.1.2. Ejemplos

##### 2.2. Automatas finitos

##### 2.2.1. Automatas finitos deterministas

#### 3. Analizadores léxicos

##### 3.1. ¿Qué es un analizador léxico?

##### 3.1.1. Importancia en el proceso de compilación

##### 3.2. Tokens, patrones y lexemas

##### 3.3. ¿Cómo funciona el analizador léxico?

#### 4. Generadores de analizadores léxicos

##### 4.1. ¿Qué son los generadores de analizadores léxicos?

##### 4.2. Generadores populares de analizadores léxicos

##### 4.2.1. FLEX

##### 4.2.1.1. Funcionamiento

##### 4.2.2. ANTLR

##### 4.2.2.1. Funcionamiento

##### 4.2.3. LEX

##### 4.2.3.1. Funcionamiento

##### 4.2.3.2. Sintaxis de reglas

1

1

2

5

6

7

9

9

11

12

12

13

16

18

20

20

22

23

23

25

25

25

27

28

29

4.3. Comparación de generadores con términos de características	30
5. Diseño y construcción de analizadores léxicos	31
5.1. Estructura de un analizador léxico generado	31
5.1.1. Concordancia de patrones basados en autómatas finitos no deterministas	32
5.1.2. Autómatas finitos deterministas para analizadores léxicos	33
5.2. Procesamiento de identificadores y literales	33
5.3. Recuperación de errores léxicos	35
5.4. Ejemplos de reglas léxicas y patrones	36
6. Aplicaciones de los analizadores léxicos	37
Conclusión	38
Bibliografía	

# ÍNDICE DE FIGURAS

Figura 1. Traducción de un compilador	1
Figura 2. Fases de un compilador	4
Figura 3. Elementos de un diagrama de transición	10
Figura 4. Ejemplo de un automata finito	10
Figura 5. Ejemplo de un AFD	11
Figura 6. Proceso de análisis léxico	13
Figura 7. División de análisis léxico y sintáctico	15
Figura 8. Ejemplos de tokens, lexemas y patrones	17
Figura 9. Especificación Flex	24
Figura 10. Árbol de expresiones	26
Figura 11. Generación de un analizador léxico	27
Figura 12. Comparación entre generadores de analizadores léxicos	30
Figura 13. Ejemplo de un AFND	32

# III Introducción

Autor: M. C. [Signature]

Al hablar del complejo campo de la compilación de lenguajes de programación se encuentra un componente importante conocido como el analizador léxico. Este elemento es el responsable de realizar un escaneo minucioso del código fuente, identificando y clasificando los elementos léxicos en unidades coherentes conocidas como tokens. Sin embargo, lo que ha revolucionado verdaderamente la creación de analizadores léxicos eficientes es la existencia de generadores específicos para esta tarea.

En el transcurso de esta monografía, se va a detallar partes importantes para comprender el contexto que rodea a los generadores de analizadores léxicos. Comenzando con el entendimiento de su esencia y funcionamiento, explorando su papel esencial en el proceso de compilación y su capacidad para simplificar la creación de analizadores léxicos que difieren por su eficiencia y exactitud.

Adicionalmente, se busca abarcar la existencia de los generadores de analizadores léxicos que se usan más, comparando algunas características. Finalizando con una demostración práctica haciendo uso de yacc y comprender en mayor profundidad el comportamiento e importancia de este tipo de herramientas.

## GENERALIDADES

El propósito de esta investigación es explorar y analizar en profundidad el campo de los generadores de analizadores léxicos en el contexto de la programación informática. Se busca comprender su importancia, funcionamiento y aplicaciones en el desarrollo del software, así como su contribución a la eficiencia y confiabilidad del proceso de compilación y análisis de programas.

El tema central de la investigación son los generadores de analizadores léxicos, herramientas cruciales en la construcción de compiladores y analizadores de código fuente. Se investigará su estructura, operación y utilidad en la identificación y clasificación de tokens y lexemas.

Se abordarán los conceptos básicos de los generadores de analizadores léxicos hasta algunos aplicaciones que estos tienen. Se analizarán ejemplos concretos y se explorará su relevancia en la detección de errores de programación.

# 1. COMPILEDORES

Los compiladores desempeñan un papel fundamental en la informática actual al funcionar como traductores que convierten lenguajes de programación diseñados para personas en lenguaje de máquina que entienden las computadoras. Además de esta función de traducción, también tienen la capacidad de identificar y señalar errores evidentes cometidos por el programador. Para la mayoría de los usuarios, un compilador se ve como una utilidad que realiza una transformación sencilla, como se muestra a continuación.



Figura 1. Traducción de un compilador

Los compiladores permiten que prácticamente todos los usuarios de computadoras no tengan que preocuparse por los aspectos específicos de la máquina relacionados con el lenguaje de máquina. Por lo tanto, los compiladores hacen que los programas sean capaces de funcionar en múltiples computadoras, logrando así la portabilidad a través de una amplia gama de sistemas.

## 1.1. ESTRUCTURA DE UN COMPILEADOR

De manera general, la estructura de un compilador consta de dos partes:

- Análisis
- Sintesis

En el análisis se divide en sus componentes esenciales el programa fuente y se le aplica una estructura gramatical. Luego, se utiliza esta

estructura para generar una versión intermedia del programa fuente. En caso de que el análisis detecte que el programa tiene errores de sintaxis o problemas de significado, debe proporcionar mensajes informativos para que el usuario pueda corregirlos. Además, en esta parte se recopila información sobre el programa fuente, la guarda en una estructura de datos denominada "tabla de símbolos", que se pasa junto con la representación intermedia a la parte de síntesis.

En la parte de síntesis se crea el programa final deseado utilizando la representación intermedia y los datos almacenados en la tabla de símbolos. Generalmente, a la parte de análisis se le llama "front end", y al lado de síntesis "back end".

### 1.1.1. FASES DE UN COMPILADOR

Si se observa con mayor detallamiento el proceso de compilación, se puede notar que se desarrolla como una serie de etapas sucesivas, en cada una de las cuales se convierte una representación del programa fuente en otra.

Dado que desarrollar un compilador no es tarea sencilla, resulta beneficioso organizar el proceso de manera jerárquica. Una aproximación común implica dividir la compilación en diferentes fases con interfaces claramente definidas. Desde un punto de vista conceptual, estas fases operan de manera secuencial, aunque en la práctica, a menudo se superponen. Cada fase, excepto la primera, toma los resultados de la fase anterior como su entrada.

generalmente, se asigna la ejecutabilidad de cada fase a un módulo independiente. Algunos de estos módulos se desarrollan manualmente, mientras que otros pueden generarse a partir de especificaciones. En ocasiones, algunos de estos módulos pueden ser compartidos entre varios compiladores.

A continuación, se explicará brevemente lo que pasa en cada fase:

• **Análisis léxico:** Implica la lectura y análisis del texto del programa. El texto se coincide y se divide en tokens, donde cada token representa un elemento del lenguaje de programación, por ejemplo, un nombre de una variable, una palabra clave o un número.

• **Análisis sintáctico:** En esta fase, la lista de tokens generada por el análisis léxico se organiza en una estructura de árbol llamada "árbol de sintaxis", que refleja la estructura del programa.

• **Verificación de tipos:** Aquí se examina el árbol de sintaxis para determinar si el programa cumple con ciertos requisitos de coherencia, como determinar si se usa una variable antes de su declaración o si se utiliza de manera inapropiada en función de su tipo, como intentar usar un booleano como un puntero de función.

• **Generación de código intermedio:** El programa es traducido a un lenguaje intermedio simple e independiente de la arquitectura de la máquina.

• **Asignación de registros:** Los nombres de las variables simbólicas utilizadas en el código intermedio se asignan a números que representan registros en el código de máquina final.

• **Generación de código de máquina:** El lenguaje intermedio se convierte en código ensamblador específico para una arquitectura de computadora particular.

*A. M. C.*

• Ensamblado y enlace: El código en lenguaje ensamblador se traduce a una representación binaria y se determinan los direcciones finales de las variables, funciones, etc.

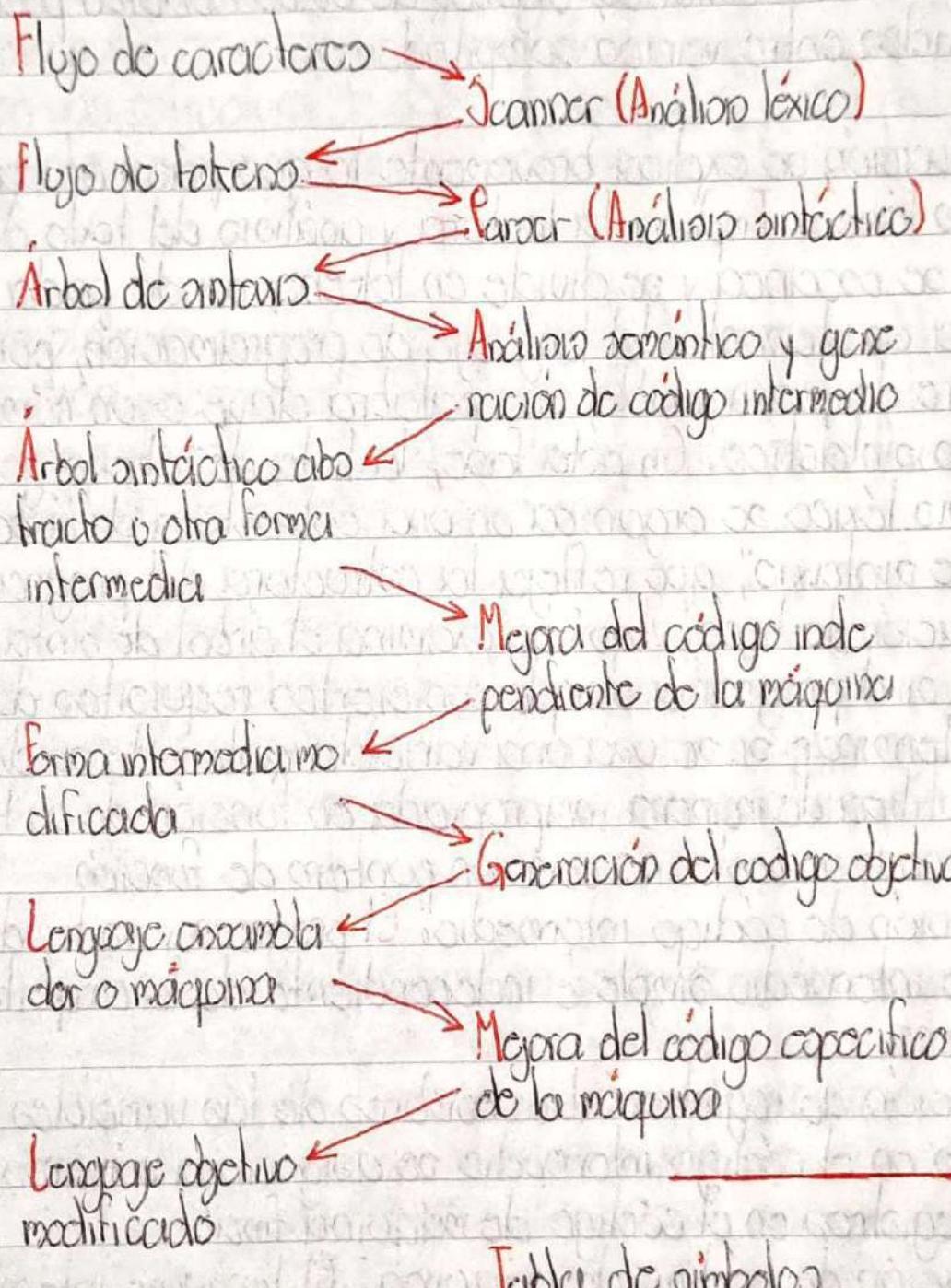


Figura 2. Fases de un compilador

## 2. ANÁLISIS LÉXICO

~~ANALISIS LÉXICO~~

a palabra léxico se refiere a lo relacionado con las palabras. En el contexto de los lenguajes de programación, las palabras son objetos como nombres de variables, números, palabras clave, etc.; que son comúnmente conocidos como tokens.

El propósito principal del análisis léxico es preparar el terreno para la fase posterior de análisis sintáctico. En teoría, el trabajo realizado durante el análisis léxico podría incorporarse directamente en el análisis sintáctico, y en sistemas simples, esto a veces sucede. Sin embargo, existen motivos para mantener estas etapas por separado.

• **Eficiencia:** Un analizador léxico realiza las tareas más simples de manera más rápida que un analizador sintáctico general. Además, dividir un sistema en dos partes puede resultar en un tamaño total más pequeño que un sistema combinado.

• **Modularidad:** La descripción sintáctica de un lenguaje no necesita abordar minuciosamente detalles léxicos como copias en blanco o comentarios.

• **Tradición:** Con frecuencia, los lenguajes se diseñan considerando frases léxicas y sintácticas separadas, y los documentos de estos lenguajes distinguen entre los elementos léxicos y sintácticos.

En el análisis léxico, se aplica emplear expresiones regulares como la metodología tradicional para definir las especificaciones. Estas expresiones son una notación algebraica utilizada para describir conjuntos de cadenas de texto.

## 2.1. EXPRESIONES REGULARES

Las expresiones regulares son una forma conveniente de especificar varios conjuntos de cadenas simples, incluso si estos conjuntos son potencialmente infinitos. Son de gran utilidad en la práctica porque permiten especificar la estructura de los tokens utilizados en un lenguaje de programación, lo que resulta especialmente útil en la programación de analizadores lógicos.

La común definición de expresión regular parte de un conjunto finito de caracteres llamado vocabulario (representado como  $\Sigma$ ). Este vocabulario puede consistir en el conjunto de caracteres utilizado por las computadoras, como el conjunto de caracteres ASCII.

En las expresiones regulares, se permite una cadena vacía o nula, que se representa como  $\lambda$ . Este símbolo denota un espacio en blanco sin contenido en el que aún no se ha encontrado una coincidencia. Además,  $\lambda$  también puede representar una parte opcional de un token. Por ejemplo, un número entero puede empezar con un signo positivo, negativo o  $\lambda$ , si es que no tiene signo.

Las cadenas de texto se crean combinando caracteres del conjunto  $\Sigma$  mediante la acción de concatenación, que implica unir caracteres individuales para formar una cadena. A medida que se unen caracteres a una cadena, ésta aumenta en longitud.

## 2.1.1. OPERADORES DE EXPRESIONES REGULARES

Un metacaracter es refiere a cualquier carácter de puntuación o operador que se utiliza en expresiones regulares. Para evitar confusión, no necesariamente encerrar un metacaracter entre comillas cuando se utiliza como un carácter común. Entre los otros símbolos que se consideran metacaracteres se encuentran: (, ), ', \*, +, !.

La alternancia ( $|$ ) puede aplicarse a conjuntos de cadenas. Si tenemos dos conjuntos de cadenas, P y Q, un string s pertenecerá a  $(P|Q)$  si y sólo si se encuentra en el conjunto P o en el conjunto Q. Por ejemplo, si LC representa el conjunto de letras mayúsculas y UC el conjunto de letras minúsculas, entonces  $(LC|UC)$  engloba todas las letras, sin importar si son mayúsculas o minúsculas.

Los conjuntos extensos o potencialmente infinitos se pueden representar de manera conveniente mediante operaciones en conjuntos finitos de caracteres y cadenas. Estas operaciones incluyen la concatenación y la alternancia. Además, se permite una tercera operación conocida como la cerradura de Kleene, que se define de la siguiente manera: El operador \* de Kleene indica que, dado un conjunto P de cadenas,  $P^*$  representa todas las cadenas que se forman al concatenar cero o más adiciones (posiblemente repetidas) de P.

Ahora que se conocen los operadores, se pueden definir las expresiones regulares de la siguiente manera:

- Ø es una expresión regular que representa el conjunto vacío, es decir, un conjunto que no contiene cadenas. Aunque raramente se

- Aprobado M. ~~Aprobado~~
- utiliza, se incluye para mantener la integridad de la notación.
  - La es una expresión regular que representa el conjunto que contiene solo la cadena vacía. **No debe confundirse con  $\emptyset$ , ya que contiene un elemento.**
  - El símbolo  $\{\alpha\}$  es una expresión regular que denota  $\{\alpha\}$ , es decir, un conjunto que contiene únicamente el símbolo  $\alpha$ , donde  $\alpha$  pertenece a  $\Sigma$ .
  - Si  $A$ , y  $B$  son expresiones regulares, entonces  $A \cup B$ ,  $AB$  y  $A^*$  también lo son.

Existen operaciones adicionales que resultan útiles en el contexto de las expresiones regulares, aunque no son estrictamente necesarias, ya que su efecto se puede lograr mediante el uso de los tres operadores principales (alternancia, concatenación y cerradura de Kleene).

- $P^+$ : A veces llamado "cierre positivo"; engloba todas las cadenas que están compuestas por una o más cadenas de  $P$  concatenadas entre si. Puede entenderse como  $P^* = (P^+ | \lambda)$ , y la notación  $P^+$  se puede expresar como  $PP^*$ .
- $\bar{A}$ : tenemos un conjunto de caracteres  $A$ .  $\text{Not}(A)$  denota el conjunto complementario de  $A$  en el conjunto  $\Sigma$ , es decir, incluye todos los caracteres de  $\Sigma$  que no estén presentes en  $A$ .
- Cuando  $k$  es una constante, el conjunto  $A^k$  representa todas las cadenas que se forman al concatenar  $k$  cadenas (posiblemente diferentes) de  $A$ . En otras palabras,  $A^k$  es equivalente a  $AAA\dots$ , con  $k$  copias de  $A$  concatenadas.

## 2.1.2. Ejemplos

1. La expresión regular  $a|b$  denota el lenguaje  $\{a, b\}$ .
2.  $(ab)(ab)$  denota  $\{aa, ab, ba, bb\}$ , el lenguaje para todos los cadenas de longitud 2 sobre el alfabeto  $\{a, b\}$ . Otra expresión regular para el mismo lenguaje es  $aabb|babb|bb$ .
3.  $a^*$  denota el lenguaje que contiene en todos los cadenas de cero o más "a", esto es,  $\{\lambda, a, aa, aaa, \dots\}$ .
4.  $(a|b)^*$  denota el conjunto de cadenas que consisten en cero o más instancias de  $a$  o  $b$ , esto es,  $\{\lambda, a, b, aa, ab, ba, bb, aaaa, \dots\}$ . Otra expresión regular para este lenguaje es  $(a^*b^*)^*$ .
5.  $a|a^*b$  denota el lenguaje  $\{a, b, ab, aab, aaab, \dots\}$ , esto es, el carácter  $a$  y todos los cadenas que comienzan en  $a$  o más "a" y terminan en  $b$ .

## 2.2. AUTÓMATAS FINITOS

Un autómata finito (**AF**) se utiliza para identificar los tokens definidos por una expresión regular. Se trata de una máquina simple y efectiva que reconoce cadenas como pertenecientes a conjuntos regulares.

- Un **AF** está compuesto por los siguientes elementos:
- Un conjunto limitado de estados.
  - Un conjunto finito de símbolos conocido como  $\Sigma$ .
  - Un conjunto de transiciones o movimientos que indican se pasa de un estado a otro, cada una de ellas etiquetada con caracteres del conjunto  $\Sigma$ .
  - Un estado especial designado como estado inicial.
  - Un subconjunto de estados que se denominan estados de aceptación o finales.

~~A~~ ~~M~~ ~~C~~

Un AF puede ser representado gráficamente mediante un diagrama de transición, el cual tiene los siguientes elementos:

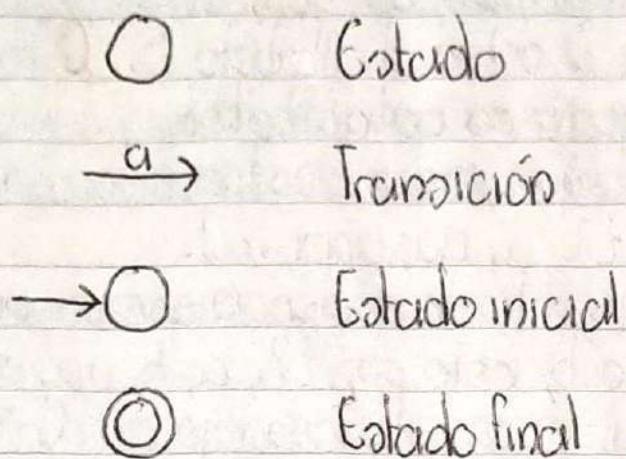


Figura 3. Elementos de un diagrama de transición

Cuando tenemos un diagrama de transición, el proceso inicia en el estado inicial. Si el siguiente carácter de entrada coincide con la etiqueta en una transición que parte del estado actual, se avanza al siguiente estado que indica la transición.

Si no hay ninguna transición posible, se detiene el proceso. Si se termina en un estado de aceptación, significa que la secuencia de caracteres que se ha leído constituye un token válido; de lo contrario, no se ha encontrado un token válido.

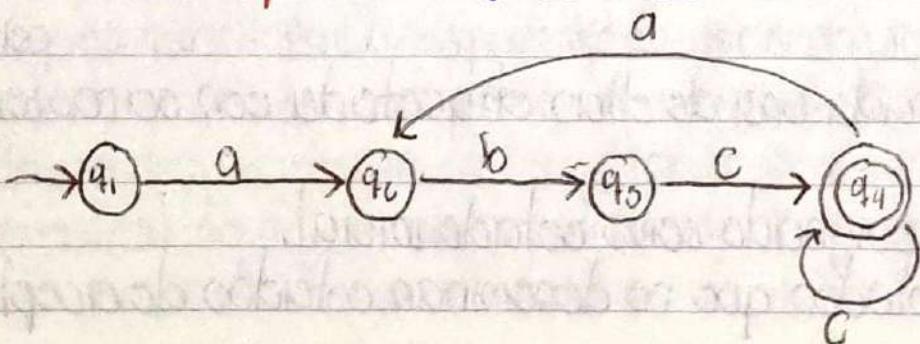


Figura 4. Ejemplo de automata finito

## 2.2.1. AUTÓMATAS FINITOS DETERMINÍSTICOS

Un Autómata Finito Determinístico (AFD) es un caso especial de Autómatas Finitos No Determinísticos (AFND) donde:

- No hay movimientos en la entrada c.
- Para cada estado s y símbolo de entrada a, hay exactamente un borde de s etiquetado como a.

Mientras que un AFND representa una abstracción de un algoritmo para identificar cadenas en un lenguaje específico, un AFD es un algoritmo más concreto y directo para cada tarea. Una gran ventaja es que cada expresión regular y cada AFND pueden convertirse en un AFD que reconozca el mismo conjunto de cadenas, ya que el AFD es realmente se utiliza cuando se crea un analizador léxico.

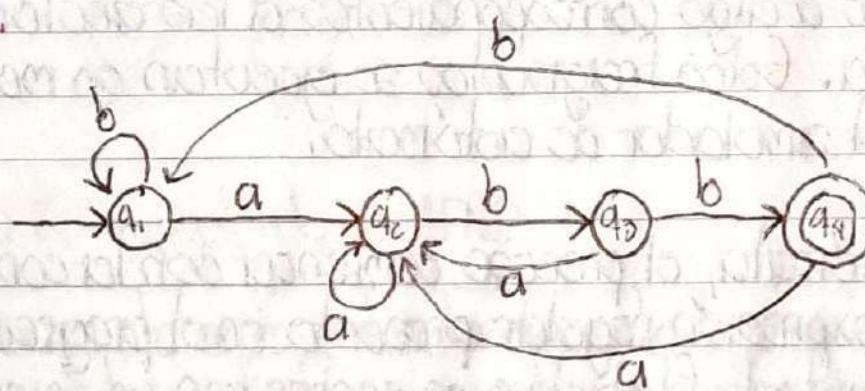


Figura 3. Ejemplo de un AFD

# EL ANALIZADOR LÉXICO

## 3.1 ¿Qué es un analizador léxico?

Un analizador léxico, también llamado "scanner" o "lexer", es una parte crucial de un compilador o intérprete de programas de computadora. Su tarea principal es examinar el código fuente de un programa y dividirlo en partes más pequeñas llamadas "tokens" o "lexemas". Estos tokens son como las piezas fundamentales del lenguaje de programación y representan palabras clave, nombres de variables, números, símbolos matemáticos y otros elementos que componen el código.

El analizador léxico trabaja siguiendo las reglas específicas del lenguaje de programación y utiliza un conjunto de reglas explícadas anteriormente que son "expresiones regulares" para conocer y categorizar los tokens en el código fuente. A medida que analiza el código, el analizador léxico genera una lista de estos tokens, cada uno etiquetado con un tipo y, en ocasiones, un valor particular. Estos tokens se pasan luego a la siguiente etapa del compilador o intérprete, que se llama "analizador sintáctico" o "parser".

para verificar la estructura gramatical del programa y entender su lógica.

Dame el siguiente componente léxico

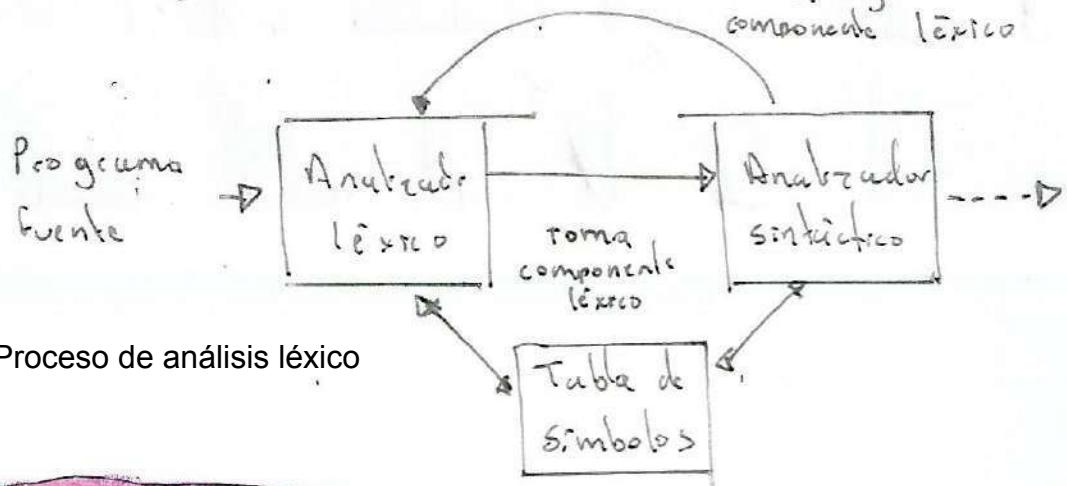


Figura 6. Proceso de análisis léxico

### 3.1.1. Importancia en el proceso de compilación

El analizador léxico desempeña un papel fundamental en la creación de programas de computadora. A continuación, se presentan cuatro puntos que abordan la importancia del mismo:

- **Reconocimiento de palabras clave:** Se puede pensar en un programa de computadora como si fuera un libro. El analizador léxico es como un lector especializado que escanea el texto y marca las palabras importantes. Estas palabras son como las piezas fundamentales del lenguaje de programación, como `if`, `while`, nombres de variables, números y signos de matemáticas.
- **Detector de errores de escritura:** A veces, cometemos errores tipográficos o escribimos algo que la computadora no entiende. El analizador léxico es como un corrector ortográfico que detecta estos errores y nos dice dónde están. Esto es útil porque podemos corregirlos antes de que causen problemas más adelante.

- **Hace el proceso más rápido:** Al imaginar que se está organizando una gran cantidad de cartas, el analizador léxico es como una máquina que las organiza rápidamente en montones según su tipo. Esto ahorra mucho tiempo, especialmente con programas largos y complejos.
- **Divide el trabajo:** En la creación de programas hay muchas etapas diferentes. El analizador léxico es como el encargado de la primera etapa, y su trabajo es diferente al de las etapas posteriores. Esto hace que el proceso sea más fácil de entender y mejorar.
- **Guarda información importante:** Cuando lees un libro, puedes tomar notas para recordar detalles importantes. El analizador léxico hace algo similar al guardar información sobre las palabras y números que encuentra. Esto es útil para las etapas siguientes del proceso de creación del programa.
- **Usa ayuda externa:** A veces, se puede obtener ayuda de herramientas especiales que hacen parte del trabajo por nosotros. En este caso, el analizador léxico puede ser creado automáticamente utilizando herramientas que conocen las reglas del lenguaje de programación. Esto facilita mucho la creación de programas.

Un aspecto relevante es entender la razón detrás de la separación entre el análisis léxico y el análisis sintáctico en lugar de optar por realizar únicamente el análisis sintáctico, aunque esto último es técnicamente posible pero poco práctico.

Hay varias razones que respaldan esta división, siendo unas de estas las que se muestran a continuación:

### LEXICO GRÁFICO / SINTÁCTICO

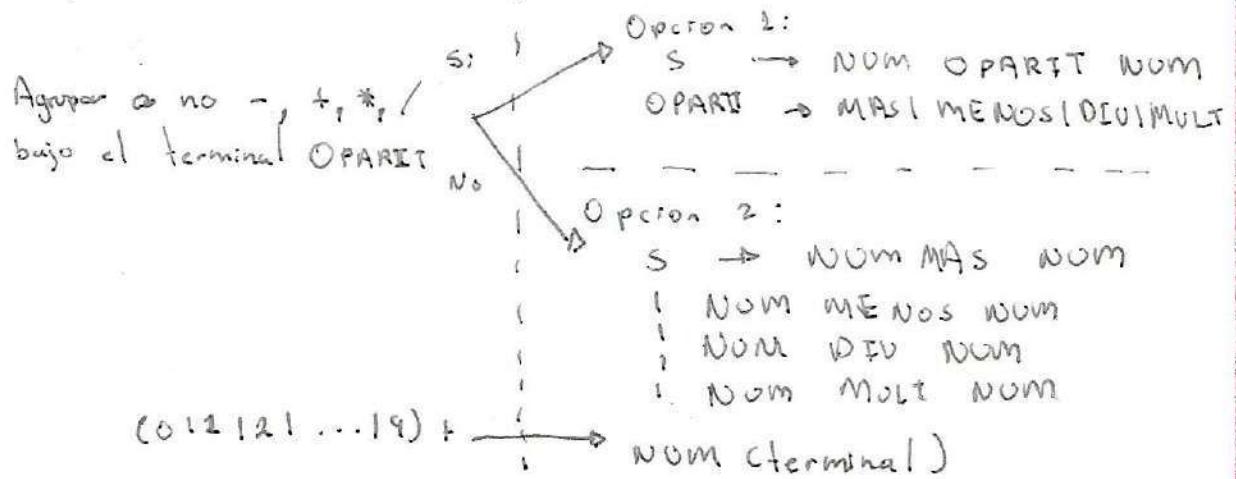


Figura 7. División de análisis léxico y sintáctico

Si el sintáctico tuviera la gramática de la "Opción 1", el lexicográfico sería:

Opción 1:  $(01121\ldots19) \rightarrow \text{NUM}$   
 $(^+^*|-^=|^*^*/^/_/^?) \rightarrow \text{OPART}$

Si en cambio el sintáctico toma la "Opción 2", el lexicográfico sería:

Opción 2:  $(01121\ldots19) \rightarrow \text{NUM}$   
 $\stackrel{+}{\sim} \rightarrow \text{MAS}$   
 $\stackrel{-}{\sim} \rightarrow \text{MENOS}$   
 $\stackrel{*}{\sim} \rightarrow \text{MULT}$   
 $\stackrel{/}{\sim} \rightarrow \text{DIV}$

Si no hubiera análisis léxico entonces, el propio analista sintáctico vería incrementado su número de reglas:

NUM  $\longrightarrow$  0  
 1  
 2  
 3  
 ...  
 1 NUM NUM

### 3.2 Tokens, patrones y lexemas

Aula M. Gómez

Dentro del análisis léxico, se utilizan tres términos interconectados:

» Un token consiste en un nombre de token y, en algunas ocasiones, un valor de atributo opcional. El nombre del token representa un tipo particular de unidad léxica en el código fuente, como una palabra clave específica o un identificador. A partir de aquí, el nombre de token se va a representar en negrita y se mencionará en muchas ocasiones por su nombre para mayor claridad.

» Por otro lado, un "patrón" se refiere a la descripción de la apariencia que pueden tener los lexemas de un token. Para tokens simples, como palabras clave, el patrón se limita a la secuencia de caracteres que compone dicha palabra clave. Sin embargo, para tokens más complejos, como los identificadores, los patrones pueden ser estructuras más elaboradas que se relacionan con múltiples cadenas de caracteres.

» Un "lexema" representa una serie de caracteres encontrados en el código fuente del programa que encaja con el patrón asociado a un token específico. Es el analizador léxico el encargado de reconocer estos lexemas como instancias de los tokens correspondientes.

A continuación se presentan unos ejemplos de tokens comunes junto con sus patrones, descritos en términos informales, así como ejemplos concretos de lexemas.

Token	Lexema	Patrón
while	while	while
Suma	+	+
Identificador	a, valor-b	[a-zA-Z]+
Número	5, 3, 25, 56	[0-9]+([0-9]+)?

Figura 8. Ejemplos de tokens, lexemas y patrones

### 3.3. ¿Cómo funciona el analizador léxico?

Como primera fase de un compilador la tarea principal del analizador léxico es leer los caracteres de entrada del programa fuente, agruparlos en lexemas y producir como salida una secuencia de tokens para cada lexema en el programa fuente. El flujo de tokens se envía al analizador para su análisis de sintaxis.

Normalmente, la interacción se implementa haciendo que el analizador llame al analizador léxico. La llamada, sugerida por el comando `getNextToken`, hace que el analizador léxico lea caracteres de su entrada hasta que pueda identificar el siguiente token que devuelve al analizador.

Dado que el analizador léxico es la parte del compilador que lee el texto fuente, puede realizar otras tareas además de la identificación de lexemas. Una de esas tareas es eliminar comentarios y espacios en blanco.

Otra tarea importante del analizador léxico es detectar errores si encuentra cosas extrañas o incorrectas en el código, como caracteres que no entiende o palabras mal escritas. En ese caso, informa al programador sobre lo que está mal para que puedan corregirlo.

Otra tarea es correlacionar los mensajes de error generados por el compilador con el programa fuente.

En algunos compiladores el analizador léxico hace una copia del programa fuente con los mensajes de error insertados en las posiciones apropiadas. Si el programa fuente utiliza un preprocessador de macros, la expansión de los macros también puede ser realizada por el analizador léxico.

Finalmente, los resultados de este análisis, los tokens que se han identificado, se pasan a otra etapa llamada "analizador sintáctico". Este último se encarga de organizar estas partes en una estructura que la computadora pueda entender, como un rompecabezas que arma la gramática del programa.

En ocasiones, los analizadores léxicos se dividen en una cascada de dos procesos:

a) El escaneo consta de procesos simples que no requieren tokenización de la entrada, como la eliminación de comentarios y la compactación de espacios en blanco consecutivos en uno.

b) El análisis léxico propiamente dicho es la parte más compleja, donde el escáner produce la secuencia de tokens como salida.

Entonces el analizador léxico utiliza estas expresiones regulares y un autómata finito para identificar y reconocer tokens en el código fuente.

# GENERADORES DE ANALIZADORES LÉXICOS

## A.1 ¿Qué son los generadores de analizadores léxicos?

Los generadores de analizadores léxicos son herramientas que hacen más fácil la creación de analizadores léxicos para lenguajes de programación o textos específicos. Estos analizadores léxicos son responsables de reconocer y categorizar partes importantes de un programa o texto, como lo son palabras clave, nombres, operadores y números.

Estos generadores simplifican la tarea de crear un analizador léxico. En lugar de escribir código manualmente, permiten definir reglas de análisis de una manera más sencilla y luego generan automáticamente el código necesario. Algunos de los generadores más conocidos son Lex, Flex y JLex, entre otros.

El proceso de construcción de un analizador léxico utilizando un generador consta de tres pasos principales:

- Definición de reglas léxicas: En esta etapa, se establecen las expresiones regulares que describen los patrones de texto que el analizador debe identificar. Estas reglas detallan cómo se identifican las partes importantes en el programa o texto.
- Generación de código: El generador de analizadores léxicos toma las reglas léxicas definidas y automáticamente crea el código fuente del analizador léxico en un lenguaje de programación específico.
- Integración y uso: Una vez que se ha generado el código, se integra en el proceso de compilación o interpretación del lenguaje en cuestión. El analizador léxico se utiliza para examinar el programa fuente o el texto de entrada y produce una secuencia de partes identificadas que se usarán en etapas posteriores del proceso de compilación o interpretación.

Los generadores de analizadores léxicos funcionan mediante la definición de reglas léxicas que describen los patrones léxicos a reconocer en un texto. Estas reglas se especifican utilizando expresiones regulares o otras formas de descripción de patrones.

## 4.2 Generadores populares de analizadores léxicos

En el proceso de analizar una cadena de texto para encontrar patrones que sean importantes, como palabras o símbolos de un lenguaje, a veces puede ser aburrido o tedioso hacerlo manualmente. Para evitar este trabajo tedioso, existen herramientas de software llamadas generadores de analizadores léxicos.

Una vez comprendidos algunos aspectos relacionados a los generadores de analizadores léxicos se puede hablar específicamente de aquellos que son más populares como los que se pueden ver a continuación:

- Flex es una herramienta ampliamente usada para crear analizadores léxicos. Ayuda a definir patrones que identifican piezas de texto en un archivo y genera un analizador para el lenguaje C.
- ANTLR es otro generador de analizadores, pero no solo léxicos, sino también sintácticos. Se destaca por crear analizadores muy eficientes y se usa en diversos proyectos de procesamiento del lenguaje.
- Lex es una herramienta clásica para crear analizadores léxicos. Fue una de las primeras en su tipo y es compatible con muchos lenguajes de programación.

#### 4.2.1 FLEX

Dentro del ámbito de los lenguajes de programación y la computación, "flex" es un término que se usa para describir una herramienta especializada. El término "Flex" es una abreviatura de "Fast lexical Analyzer Generator" (Generador Rápido de Analizadores Léxicos), y su principal función es ayudar en la creación de herramientas que puedan entender y procesar el código escrito en lenguajes de programación y otros tipos de texto estructurado.

Se utiliza para establecer las reglas de la manera en la que se detectan los tokens en un archivo de especificaciones. Posteriormente genera automáticamente el código necesario en lenguajes como C o C++, este código resultante se integra en programas más grandes, como compiladores o intérpretes, para permitirles entender y procesar el código fuente de manera efectiva.

##### 4.2.1.1. Funcionamiento

Se muestra a continuación una descripción del funcionamiento básico de Flex:

- Herramientas para construir analizadores léxicos: Flex es una herramienta que se utiliza para crear analizadores léxicos.
- Entrada de descripciones de tokens: Cada descripción consta de un patrón y una acción asociada que se ejecutará cuando se encuentre ese patrón.

- **Llamado de la función `yyflex()`:** flex toma estas descripciones de tokens y genera automáticamente una función en el lenguaje C llamada `yyflex()`. Esta función es el analizador léxico resultante y se encarga de escanear el código fuente en busca de los patrones especificados.
- **Uso de expresiones regulares:** Las descripciones de tokens se basan en patrones que son extensiones de las expresiones regulares. Las expresiones regulares son notaciones que permiten especificar de manera concisa y precisa patrones de texto, de esta manera Flex identifica tokens en el código fuente.
- **Especificación Flex:** El conjunto completo de descripciones de tokens se llama la "especificación Flex". Esta especificación es escrita para que Flex genere correctamente el analizador léxico. Contiene todas las reglas y patrones que el analizador léxico debe seguir para reconocer tokens en el código fuente.

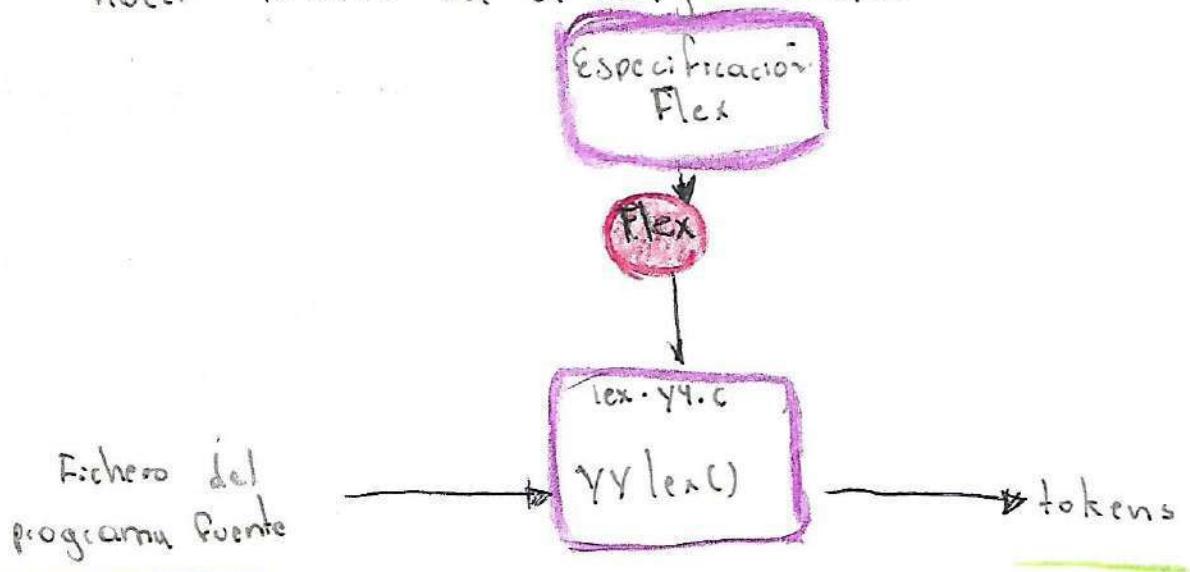


Figura 9. Especificación Flex

## 4.22. ANTLR

ANTLR es una potente herramienta que se utiliza para la creación de analizadores, que son programas capaces de convertir fragmentos de texto en estructuras organizadas, como árboles de sintaxis abstracta (AST).

La popularidad de ANTLR se debe a que cumple con varios requisitos fundamentales que los programadores buscan en sus herramientas. Estos requisitos incluyen el uso de mecanismos comprensibles, la capacidad de resolver problemas complejos, flexibilidad, automatización de tareas repetitivas y la generación de resultados que se integran fácilmente en una aplicación.

ANTLR se destaca por su sintaxis consistente para la especificación de analizadores léxicos, analizadores sintácticos y analizadores de árboles.

### 4.2.2.1 Funcionamiento

1.. Definición de reglas: Se establecen las reglas del lenguaje que se desea analizar, ANTLR tiene su propio lenguaje para escribir estas reglas.

2.. Generación de analizadores: En ANTLR hay dos tipos de herramientas: una que busca palabras especiales en el código y otra que examina cómo se organizan las partes del código.

3.. Agregar acciones personalizadas: Se pueden agregar instrucciones personalizadas para que se ejecuten cuando se encuentre cierto tipo de código.

- 4.. Análisis del código fuente: Se buscan las palabras y estructuras importantes según las reglas definidas.
- 5.. Creación de estructuras de datos: Si se han agregado acciones personalizadas, se ejecutan durante el análisis y se pueden utilizar para crear representaciones especiales de la información del código, como organizarla en un árbol de decisiones.
- 6.. Integración en una aplicación: Los resultados del análisis se pueden usar en una aplicación más grande.

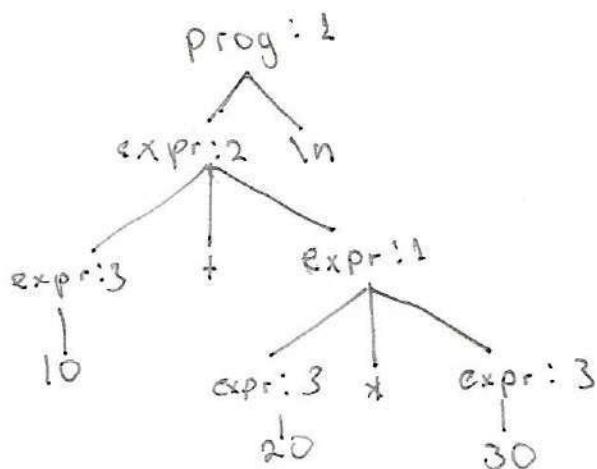


Figura 10. Árbol de expresiones

### 2.3. LEX

Lex es un generador de analizadores léxicos que fue creado para hacer esta tarea de manera más fácil. Básicamente, te permite definir cómo deberían buscarse las partes importantes en el texto, y luego genera automáticamente el código para hacerlo.

Lex produce un módulo de escáner completo, codificado en C, que puede compilarse y vincularse con otros módulos compiladores. En los programas creados, hay una función llamada "yylex()": que se encarga de buscar palabras en un texto.

"yylex()": es parte de algunos generadores mencionados con anterioridad, lo interesante es que es muy adaptable, lo que significa que puede hacer diferentes cosas según lo que se busque.

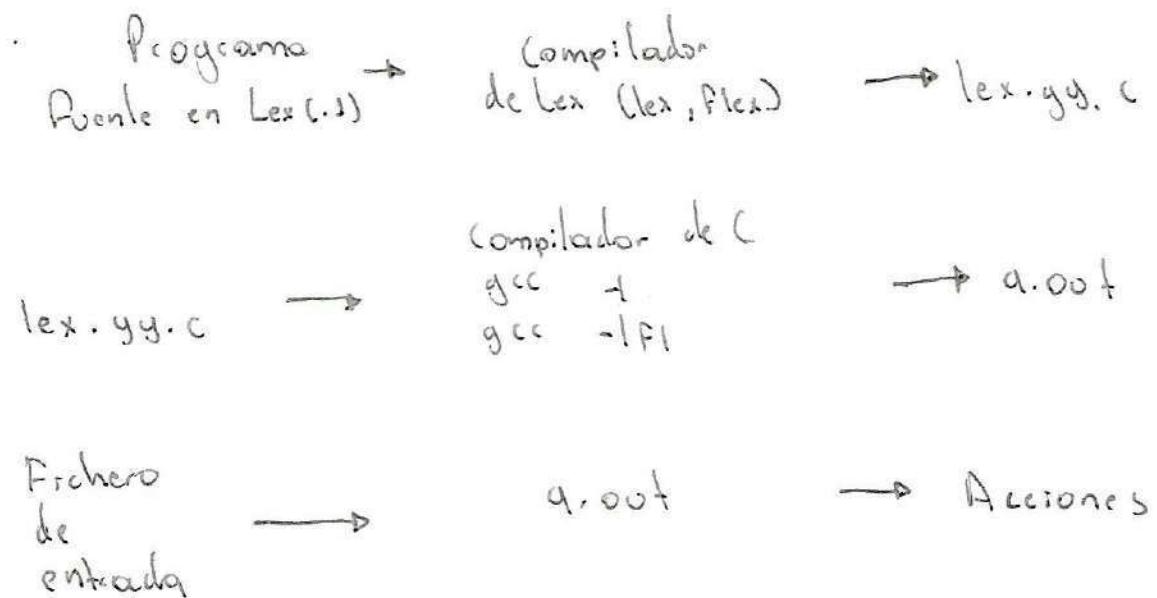


Figura 11. Generación de un analizador léxico

#### 4.2.3.1 Funcionamiento

Un programa en Lex sigue una estructura típica que consta de tres secciones. La segunda opción es obligatoria, por lo que un programa Lex podría verse así:

<sección de definiciones>

% %

<sección de reglas>

% %

<sección de rutinas>

La primera sección incluye declaraciones de variables, constantes y definiciones regulares que facilitan el uso de expresiones regulares largas en la sección de reglas. Por ejemplo:

letra [A-Za-z]

En la segunda sección se especifican los patrones que se desean reconocer y las acciones que deben llevarse a cabo.

La tercera sección permite definir funciones auxiliares en C. El programa generado por Lex, que tiene por defecto el nombre a.out (a menos que se especifique otro), tiene el siguiente comportamiento:

- Imprime en la salida estándar los lexemas que no coinciden con ninguno de los patrones especificados.
- Ejecuta la acción indicada para los lexemas que coinciden con alguno de los patrones definidos.

#### 4.2.3.2. Sección de reglas.

Un programa en Lex tiene una estructura básica que se compone de dos partes:

patrón - regular    acción - en - C

El patrón regular que debe ubicarse al principio de la linea sin espacios del patrón. Lex admite expresiones regulares en formato similar al de "egrep" con algunas adicionales detalladas en la documentación.

Existen acciones especiales en Lex como "%ECHO", que copia el lexema reconocido en la salida estandar, equivalente a usar printf ("%s", ytext). También se utiliza el símbolo "%:" para indicar que, para este patrón, se debe ejecutar la acción correspondiente al patrón inmediatamente inferior.

### 4.3 Comparación de generadores en términos de características y usos.

Comparar diferentes generadores de analizadores léxicos y sintácticos, como Flex, ANTLR y Lex, es una práctica importante en el mundo de la programación. Esto se debe a que cada uno de estos generadores tiene sus propias características y habilidades únicas. Esto ayuda a tomar decisiones informadas y a elegir la herramienta que mejor se adapte a lo que busca realizar. Visualizando estas comparaciones en el siguiente cuadro comparativo:

Características	Flex	ANTLR	Lex
Lenguaje de programación soportado	C/C++	Java	C
Gramática soportada	Expresiones regulares	Gramáticas libres de contexto	Expresiones regulares
Nivel de abstracción	Bajo	Alto	Bajo
Compatibilidad multiplataforma	Cuenta con compatibilidad	Cuenta con compatibilidad	Cuenta con compatibilidad
Soporte para gramáticos	Limitado	Amplio	Limitado
Características avanzadas	Limitadas	Sí (Herencia, Árboles de Sintaxis)	Limitadas

Figura 12. Comparación entre generadores de analizadores léxicos

## D. DISEÑO Y CONSTRUCCIÓN DE ANALIZADORES LÉXICOS

### 5.1. ESTRUCTURA DE UN ANALIZADOR LÉXICO GENERADO

Un analizador léxico generado por Lex tiene una estructura específica. Consiste en un programa base que simula un autómata, sin especificar si es determinista o no determinista. El resto del analizador léxico se compone de elementos generados automáticamente por Lex a partir del programa que se le proporciona.

Estos elementos incluyen:

- 1.- Una tabla de transición que define el comportamiento del autómata.
- 2.- Funciones que son incorporadas directamente en el código generado por Lex.
- 3.- Fragmentos de código correspondientes a las acciones del programa de entrada. Estos fragmentos se ejecutan en momentos específicos por el simulador de autómata.

Para crear el autómata, el proceso comienza con la conversión de cada patrón de expresión regular presente en el programa Lex en un AFND individual. El objetivo es contar con un único autómata que sea capaz de reconocer lexemas que coincidan con cualquiera de los patrones definidos en el programa. Para lograr esto, se fusionan todos los AFND en un solo autómata introduciendo un nuevo estado inicial y estableciendo transiciones desde este nuevo estado hacia los estados iniciales de cada uno de los AFND correspondientes a los patrones individuales.

### 5.1.1. CONCORDANCIA DE PATRONES BASADO EN AFND

Si el analizador lógico está trabajando en un **AFND**, su proceso de lectura de la entrada se inicia desde el punto al que se hace referencia como "lexemaBegin". A medida que avanza el trazo de la entrada, y move el puntero denominado "hacia adelante", determina el conjunto de estados en los que se encuentra en cada punto durante este proceso.

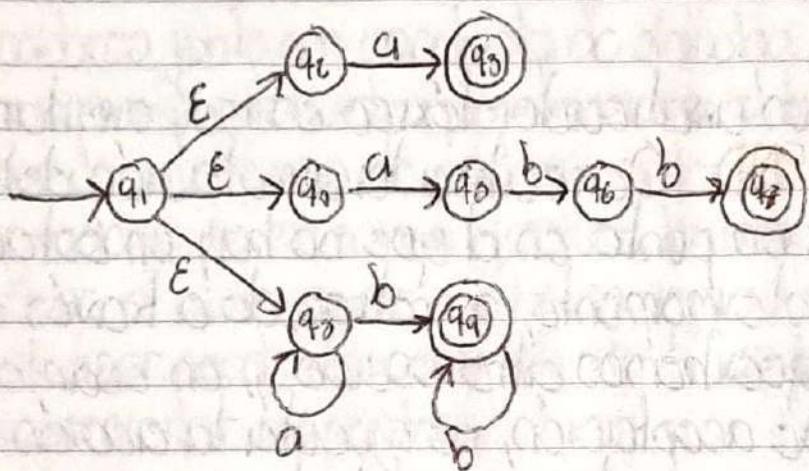


Figura 13. Ejemplo de un AFND

Al llegar a un punto en la entrada durante la simulación del **AFND** en el que ya no hay estados siguientes, se reconoce que no existe la posibilidad de que un prefijo adicional de la entrada conduzca al **AFND** a un estado de aceptación. En su lugar, el conjunto de estados siempre permanecerá vacío. Por lo tanto, en este momento se está en condiciones de determinar cuál es el prefijo más largo de la entrada que corresponde a un lexema que coincide con alguno de los patrones definidos.

### 3.1.2. AFD PARA ANALIZADORES LÉXICOS

Otra estructura implica la conversión del AFND correspondiente a todos los patrones en un AFD equivalente utilizando la técnica de construcción de subconjuntos. Dentro de cada estado del AFD, si se encuentran uno o más estados de aceptación del AFND, se determina cuál es el primer patrón cuyo estado de aceptación está representado en el AFD y se asigna ese patrón como la acción del AFD.

La utilización del AFD en un analizador léxico es muy similar a como se hace con el AFND. Se realiza la simulación del AFD hasta que se llega a un punto en el que no hay un estado siguiente disponible. En ese momento, se retrocede a través de la secuencia de estados que se han atravesado y, en cuanto se encuentra un estado de aceptación, se ejecuta la acción asociada al patrón que corresponde a ese estado.

### 5.2. PROCEDIMIENTO DE IDENTIFICADORES Y LITERALGO

En lenguajes de programación simples que sólo incluyen variables y declaraciones globales, el compilador por lo general agrega un identificador de inmediato a una tabla de símbolos si éste no está presente en ella. Tanto si el identificador se agrega como si ya estaba en la tabla, el compilador proporciona un puntero a la entrada de la tabla de símbolos correspondientes.

En lenguajes estructurados en bloques, la situación es diferente ya que un mismo identificador puede utilizarse en diversos contextos, como variable, miembro de una clase o etiqueta. En estos

caso, el compilador generalmente no puede determinar cuándo agregar un identificador a la lista de símbolos para el ámbito actual o cuándo debe regresar un puntero a una instancia de un ámbito previo. Algunos compiladores copian el identificador en una variable de cadena privada que no se sobrecarga y devuelven un puntero.

En ocasiones, se utiliza un espacio de cadena para almacenar los identificadores junto con una tabla de símbolos. Este espacio de cadena es una memoria extensible que almacena el texto de los identificadores. El uso de un espacio de cadena evita las llamadas frecuentes a los asignadores de memoria y reduce la sobrecarga de copiado que se produce al almacenar múltiples copias del mismo identificador. El compilador puede agregar un identificador al espacio de cadena y proporcionar un puntero.

Otra alternativa al espacio de cadena es una tabla hash que almacena identificadores y les asigna un número de serie único. Este número de serie es un número entero pequeño que puede utilizarse en lugar de un puntero de espacio de cadena. Los identificadores con el mismo texto obtienen el mismo número de serie, mientras que los identificadores con textos diferentes obtienen números de serie diferentes.

Para otros tipos, como los literales, se requiere un procesamiento adicional antes de devolverlos.

### 3.3. RECUPERACIÓN DE ERRORES LÉXICOS

Cuando se encuentra una secuencia de caracteres que no se puede convertir en un token válido, se produce un error léxico. Aunque estos errores son poco comunes, es necesario abordarlos en el escáner. No tiene sentido interrumpir el proceso de compilación debido a lo que generalmente es un error menor, por lo que normalmente se busca alguna forma de recuperación de errores léxicos. Puede haber dos enfoques:

- Descartar los caracteres leídos hasta ese momento y reiniciar la exploración a partir del siguiente carácter no leído.
- Descartar el primer carácter que leyo el escáner y reanudar la exploración desde el siguiente carácter.

En la mayoría de los casos, un error léxico se debe a la presencia de algún carácter no válido, que suele ser el inicio de un token. En esta situación, ambos enfoques funcionan igual de bien. Sin embargo, en situaciones donde un buen algoritmo de reparación de errores sintácticos puede realizar una corrección razonable, podría ser útil devolver un token de advertencia especial cuando se encuentra un error léxico. El valor numérico de este token de advertencia es la cadena de caracteres que se ha eliminado para reiniciar el escáner. El token de advertencia informa al analizador de que el siguiente token puede no ser confiable y que podría ser necesario corregir el error.

## 5.1. EJEMPLOS DE REGLAS LÉXICAS Y PATRÓN

A continuación, se muestran algunos ejemplos de reglas léxicas y patrón en un analizador léxico.

### • Identificadores

Regla: Deben comenzar con una letra y puede estar seguido por letras, números o guioncilla bajos.

Patrón: [a-zA-Z][a-zA-Z0-9\_]\*

### • Números enteros

Regla: Una secuencia de dígitos

Patrón: [0-9]\*

### • Números decimales

Regla: Constán de una parte entera seguida de un punto decimal y una parte fraccional, ambas partes pueden estar vacías.

Patrón: [0-9]\*\.[0-9]+ | [0-9]+\.[0-9]\*

### • Palabras clave

Reglas: Son palabras reservadas con un significado especial.

Patrón: if | else | while | for | function | return | ...

### • Comentarios

Regla: Texto que se ignora durante la compilación/interpretación

Patrón (comentarios de una línea): // \*

Patrón (comentarios multilínea): /\* ([\*]|\\*+[\*/]\*)\*/ + /

# 6. APLICACIONES DE LOS ANALIZADORES LÉXICOS

Además de para construir compiladores e intérpretes, los analizadores léxicos se pueden emplear para muchos programas "convencionales". Un analizador lógico simplifica notablemente la interfaz y si se dispone de un generador automático, el problema se resuelve en pocas líneas de código.

Entre otras aplicaciones se incluyen:

- o Resaltado de sintaxis: En editores de texto y entornos de desarrollo integrados (IDE), esto simplifica la labor de los programadores al identificar elementos como palabras clave, nombres de variables, cadenas de texto, y números.
- o Examen de código estático: Se usan estas herramientas para descubrir problemas potenciales en el código fuente.
- o Mejora de código: Desempeñan un papel en el proceso de mejorar el rendimiento del código, al detectar patrones específicos en el mismo.
- o Comprensión de lenguaje natural: Se usa para dividir un texto en palabras y oraciones.
- o Análisis de registros y registro de eventos: Se emplea para analizar registros de eventos con el fin de identificar patrones de comportamiento o problemas en sistemas complejos.

- 10/10
- o Generación automática de documentación: Los utilizan para extraer información de los comentarios en el código fuente y así generar automáticamente documentación o asistencia para los desarrolladores.
  - o Exploración de datos: Pueden usarse para analizar archivos de datos y extraer información relevante.
  - o Procesamiento de consultas: Pueden ser empleados para procesar consultas de usuario y convertirlas en comandos que el sistema pueda comprender.
  - o Seguridad informática: En el ámbito de la seguridad informática, se usan generadores para examinar el tráfico de red en busca de patrones que puedan indicar ataques o comportamientos maliciosos.

## || Conclusión ||

En esta monografía se ha explorado minuciosamente el papel esencial desempeñado por los generadores de analizadores léxicos en el tema de la compilación. Primariamente se ha analizado el funcionamiento del analizador léxico, que se encarga de escanear y reconocer los elementos léxicos del código fuente, transformandolos en tokens coherentes.

La relevancia de los generadores de analizadores léxicos radica en su capacidad para acelerar el desarrollo de compiladores, lo que conlleva a una reducción del riesgo de errores humanos y a un análisis léxico más preciso. Además, se ha llevado a cabo una evaluación comparativa de diversos generadores disponibles en el mercado, lo que proporciona una base sólida para la elección adecuada de una herramienta según las necesidades particulares de trabajo de compilación. Finalizando con una realización de manera satisfactoria de un ejercicio práctico.



## Bibliografía

UNIVERSIDAD EUROPEA DE MADRID. (s. f.). ANÁLISIS LÉXICO. Cartagena99. Recuperado 21 de septiembre de 2023, de

[https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2\\_M4\\_U2\\_T1.pdf](https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2_M4_U2_T1.pdf)

Bavera, F., Nordio, D., & Arroyo, M. (s. f.). JTLEX un Generador de Analizadores Léxicos Traductores. sedici. Recuperado 21 de septiembre de 2023, de

[http://sedici.unlp.edu.ar/bitstream/handle/10915/23090/Documento\\_completo.pdf?sequence=1&isAllowed=y#:~:text=Un%20analizador%20l%C3%A9xico%20es%20un,tokens%20correspondientes%20y%20sus%20atributos.](http://sedici.unlp.edu.ar/bitstream/handle/10915/23090/Documento_completo.pdf?sequence=1&isAllowed=y#:~:text=Un%20analizador%20l%C3%A9xico%20es%20un,tokens%20correspondientes%20y%20sus%20atributos.)

[Agu01] J. Aguirre, V. Grinspan, M. Arroyo, J. Felippa, G. Gomez, "JACC un entorno de generación de procesadores de lenguajes" Anales del CACIQ 01, 2001.

Ortega, A., & De la Cruz, M. (2008). Construcción de un analizador léxico para ALFA con Flex. arantxa. Recuperado 21 de septiembre de 2023, de

[http://arantxa.ii.uam.es/~mdlcrus/docencia/compiladores/2007\\_2008/lexico\\_07\\_08.pdf](http://arantxa.ii.uam.es/~mdlcrus/docencia/compiladores/2007_2008/lexico_07_08.pdf)

Hernandez, G. S. Q. (s. f.). GENERADOR DE ANALIZADORES LÉXICOS (LEX & FLEX). prezi.com.

<https://prezi.com/p/skykfvgirxm7/generador-de-analizadores-lexicos-lex-flex/>

De València Escola Tècnica Superior d'Enginyeria Informàtica, U. P. (2020, 9 noviembre). Flex. Desarrollo de un analizador léxico usando flex. <https://riunet.upv.es/handle/10251/10071>

Parr, T. J., & Quong, R. W. (1995). ANTLR: A predicated-LL(K) Parser generator. Software - Practice and Experience, 25(7), 789-810. <https://doi.org/10.1002/spe.4380250705>

Bavera, F. (2002, 1 octubre). JTLEX un generador de analizadores léxicos traductores.

<http://sedici.unlp.edu.ar/handle/10915/23090>

Scott, M. L. (2006). *Programming Language Pragmatics* (2<sup>a</sup> ed.).

Elsevier. <https://doi.org/10.1016/b978-0-12-374514-9.x0001-8> (Obra original publicada en 2000)

Mogensen, T. (2007). *Basics of compiler design* (2<sup>a</sup> ed.).

s.n.]. <http://hjemmesider.diku.dk/~torbenm/Basics/> (Obra original publicada en 2000)

Levine, J., Brown, D., & Mason, T. (1992). *lex & yacc* (2<sup>a</sup> ed.). O'Reilly

Media. <http://www.nylxs.com/docs/lexandyacc.pdf>

Fischer, C. N., Cytron, R. K., & LeBlanc Jr, R. J. (2010). *Crafting a compiler*. Pearson

Education. <https://studylib.net/doc/26053844/crafting-a-compiler-by-charles-n.-fischer--ron-k.-cytron-...>

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers, principles, techniques, and tools* (2<sup>a</sup> ed.). Pearson

Education. <https://drive.google.com/file/d/0B1MogsyNAsj9eIVzQWR5NWVTSVE/view?resourcekey=0-zoBDMpzTafr6toxDuQLNUg> (Obra original publicada en 1986)

Two handwritten signatures are present at the top right of the page. The first signature appears to be 'J. M. Vilar' with a small checkmark next to it. The second signature is more stylized and less legible.

Vilar Torres, J. M. (2010, October 7). *Analizador léxico*. Universitat Jaume I.  
[https://repositori.uji.es/xmlui/bitstream/handle/10234/22657/II26\\_analisis\\_lexico.pdf  
?sequence=1](https://repositori.uji.es/xmlui/bitstream/handle/10234/22657/II26_analisis_lexico.pdf?sequence=1)

TECNOLÓGICO  
NACIONAL DE MÉXICO**INSTITUTO TECNOLÓGICO DE CELAYA****PRÁCTICA DE LABORATORIO - LENGUAJES  
Y AUTÓMATAS II****AUTORES:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

CARRERA	NOMBRE DE LA ASIGNATURA
INGENIERIA EN SISTEMAS COMPUTACIONALES	LENGUAJES Y AUTÓMATAS II

PRACTICA No.	NOMBRE DE LA PRACTICA	FECHA DE ENTREGA
1	Aplicaciones de los generadores de analizadores léxicos	27/09/2023

1	INTRODUCCION
En esta práctica, se empleará una herramienta de software de código abierto denominada Analizador Léxico FLEX para desarrollar un ejercicio de demostración con el propósito de esclarecer y ejemplificar al menos tres definiciones léxicas específicas del lenguaje de programación C. A través de esta actividad, se investigarán los principios fundamentales que norman la detección de palabras clave, operadores y otros elementos primordiales, sentando así una base sólida para futuros desarrollos y una comprensión más profunda de este lenguaje de programación.	

2	OBJETIVO
El objetivo de la práctica es permitir la comprensión práctica del análisis léxico través de la creación y demostración de ejemplos concretos de definiciones léxicas utilizando el Analizador Léxico FLEX, con el fin de fortalecer el conocimiento sobre el proceso de análisis léxico, las definiciones léxicas y, a su vez, adquirir habilidades en este tipo de herramientas.	

3	FUNDAMENTO
<p>En un compilador, el análisis léxico es la primera fase en el proceso de compilación. Esta fase consiste en analizar el código fuente de un programa para dividirlo en una secuencia de tokens. Estos tokens representan elementos de un lenguaje de programación como palabras clave, identificadores, operadores, constantes, entre otras cosas.</p> <p>El objetivo principal del análisis léxico es facilitar el procesamiento del código fuente cuando este pase a etapas posteriores del compilador. Con la división del código en tokens, se proporciona una representación más estructurada y comprensible para las otras etapas del compilador.</p> <p>Las expresiones regulares son patrones que se utilizan para poder buscar texto en un documento o cadena de caracteres. Estas ayudan a definir un conjunto de reglas de búsqueda.</p> <p>Las expresiones regulares son ampliamente utilizadas en la programación y en el procesamiento de textos para realizar tareas como validaciones, búsqueda y reemplazo de patrones específicos, análisis de texto, etc. Permiten realizar búsquedas complejas y flexibles para buscar una coincidencia exacta en un texto.</p> <p>Un analizador léxico es una herramienta que utilizan los compiladores para analizar y reconocer elementos léxicos del código fuente de un programa. El analizador léxico lee los caracteres de</p>	

TECNOLÓGICO  
NACIONAL DE MÉXICO**INSTITUTO TECNOLÓGICO DE CELAYA****PRÁCTICA DE LABORATORIO - LENGUAJES  
Y AUTÓMATAS II****AUTORES:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

entrada que contiene el código fuente y, después, genera como salida una secuencia de tokens. Estos tokens representan elementos del lenguaje de programación, como palabras reservadas, identificadores, operadores, etc.

Los analizadores léxicos también utilizan expresiones regulares para poder definir los patrones de búsqueda y reconocer los tokens. Las expresiones regulares van a describir las reglas sintácticas del lenguaje de programación y van a permitir identificar y clasificar los elementos léxicos del código fuente.

Un generador de analizadores léxicos es una herramienta que permite automatizar la creación de analizadores léxicos. Estas herramientas toman como entrada una especificación de reglas léxicas y, posteriormente, generan automáticamente el código fuente del analizador léxico.

Estos generadores proporcionan una gran cantidad de ventajas como rapidez en el desarrollo, facilidad de mantenimiento y reducción de errores en la implementación. También permiten separar la responsabilidad del análisis léxico del resto del compilador, lo que permite la modularidad y escalabilidad del sistema.

Algunos ejemplos de generadores de analizadores léxicos son Lex, Flex, JLex y AT&T Lex.

Flex es un generador de analizadores léxicos que se utiliza en el desarrollo de compiladores y lenguajes de programación. Toma como entrada las especificaciones de las reglas léxicas utilizando expresiones regulares y, después, genera el código fuente del analizador léxico en el lenguaje de programación deseado, como C.

El analizador léxico que genera Flex puede reconocer y dividir el código fuente en tokens, luego, utiliza las reglas que se especificaron para poder identificar y clasificar estos tokens. También puede manejar expresiones regulares extendidas, reconocer patrones regulares complejos y proporcionar acciones asociadas a cada token.

Otra característica importante de Flex es que es compatible con distintos sistemas operativos y se puede integrar con otras herramientas de compilación, como Yacc, un generador de analizadores sintácticos.

El lenguaje de programación C es un lenguaje de alto nivel y de propósito general. Es conocido por su eficiencia y porque es capaz de acceder directamente a la memoria y otros dispositivos periféricos. Además, ofrece una gran variedad de características como tipos de datos estáticos, estructuras de control. Funciones, punteros y arreglos.

Este lenguaje es ampliamente utilizado gracias a su eficiencia, portabilidad y flexibilidad; ha sido utilizado como base para el desarrollo de otros lenguajes de programación como C++ y C#, también se usa para implementar sistemas operativos, compiladores, aplicaciones embebidas y otros tipos de software.

4

**MATERIALES NECESARIOS**

- Computadora.
- Máquina virtual con la distribución de Linux de su preferencia.
- Editor de texto en la distribución de Linux
- Editor de código (opcional)

TECNOLÓGICO  
NACIONAL DE MÉXICO**INSTITUTO TECNOLÓGICO DE CELAYA****PRÁCTICA DE LABORATORIO - LENGUAJES  
Y AUTÓMATAS II****AUTORES:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

**5****DESARROLLO****Algoritmo**

1. Definir las variables necesarias, incluyendo expresión (la cadena de entrada), token (para almacenar los tokens reconocidos), i (un índice para recorrer la cadena), carácter (para almacenar el carácter actual), y varias banderas (esIdentificador, esNúmeroEntero, esDeclaracionVariable, esEstructuraDecision, esEspacioEnBlanco) para rastrear el tipo de elemento reconocido.
2. Solicitar al usuario ingresar una expresión.
3. Inicializa i en 1 y comienza un bucle while para recorrer la cadena expresión.
4. En cada iteración del bucle, examinar el carácter actual.
5. Si el carácter es una letra, un guión bajo o un dígito, se considera que comienza un posible identificador. Se recopila el carácter en token y se continúa recorriendo la cadena hasta que se encuentre un carácter que no sea parte de un identificador. La bandera esIdentificador se establece en Verdadero.
6. Si el carácter es un signo más o un signo menos, se considera que comienza un posible número entero. Se recopila el carácter en token y se continúa recorriendo la cadena hasta que se encuentre un carácter que no sea un dígito. La bandera esNúmeroEntero se establece en Verdadero.
7. Si se detecta un espacio en blanco, se verifica si pudiera ser el inicio de una declaración de variable. Se recopilan los caracteres hasta encontrar un = seguido de ;. Si es así, la bandera esDeclaracionVariable se establece en Verdadero.
8. Si se detecta la palabra "if" seguida de un paréntesis (, se verifica si pudiera ser una estructura de decisión "if". Se recopilan los caracteres hasta encontrar un paréntesis de cierre ) seguido de una llave {. Si es así, la bandera esEstructuraDecision se establece en Verdadero.
9. Verificar si el carácter es un espacio en blanco, una tabulación o un salto de línea y, si es así, establecer la bandera esEspacioEnBlanco en Verdadero.
10. Imprimir los resultados identificados junto con el tipo de elemento reconocido (identificador, número entero, declaración de variable, estructura de decisión) según las banderas establecidas.
11. Restablecer todas las banderas antes de procesar el siguiente carácter.

**Pseudocódigo****Algoritmo AnalizadorLexico**

Definir expresion Como Cadena  
Definir token Como Cadena  
Definir i Como Entero  
Definir caracter Como Caracter  
Definir esIdentificador Como Logico  
Definir esNúmeroEntero Como Logico  
Definir esDeclaracionVariable Como Logico  
Definir esEstructuraDecision Como Logico  
Definir esEspacioEnBlanco Como Logico



TECNOLÓGICO  
NACIONAL DE MÉXICO

## INSTITUTO TECNOLÓGICO DE CELAYA



### PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

#### AUTORES:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

*[Handwritten signatures]*

Escribir "Ingrese una expresión: "

Leer expresion

i <- 1

Mientras i <= Longitud(expresion) Hacer  
    caracter <- Subcadena(expresion, i, 1)  
    token <- ""

// Regla para identificadores  
Si expresion=caracter O caracter = "\_" Entonces  
    esIdentificador <- Verdadero  
Mientras expresion=caracter O expresion=dígito O caracter = "\_" Hacer  
    token <- token + caracter  
    i <- i + 1  
    Si i <= Longitud(expresion) Entonces  
        caracter <- Subcadena(expresion, i, 1)  
    Sino  
        // Fin de la cadena  
    Fin Si  
Fin Mientras  
Fin Si  
    Fin Mientras

// Regla para números enteros  
Si carácter = "+" O carácter = "-" Entonces  
    token <- token + carácter  
    i <- i + 1  
    caracter <- Subcadena(expresion, i, 1)  
Fin Si

Mientras expresion=caracter Hacer  
    token <- token + caracter  
    i <- i + 1  
    Si i <= Longitud(expresion) Entonces  
        caracter <- Subcadena(expresion, i, 1)  
    Sino  
        // Fin de la cadena  
    Fin Si



TECNOLÓGICO  
NACIONAL DE MÉXICO

## INSTITUTO TECNOLÓGICO DE CELAYA



### PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

#### AUTORES:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

```
Si Longitud(token) > 0 Entonces
    esNúmeroEntero <- Verdadero
Fin Si

// Regla para declaraciones de variables
Si token = " " Entonces
    token <- ""
    Mientras expresion=cadena O caracter = "_" Hacer
        token <- token + caracter
        i <- i + 1
    Si i <= Longitud(expresion) Entonces
        caracter <- Subcadena(expresion, i, 1)
    Sino
        // Fin de la cadena
    Fin Si
Fin Mientras
Si caracter = " " Y Subcadena(expresion, i + 1, 1) = "=" Y Subcadena(expresion, i + 3, 1) = ";""
Entonces
    esDeclaracionVariable <- Verdadero
Fin Si
Fin Si

// Regla para estructuras de decisión "if"
Si token = "if" Y caracter = "(" Entonces
    token <- ""
    Mientras caracter <> ")" Hacer
        token <- token + caracter
        i <- i + 1
    Si i <= Longitud(expresion) Entonces
        caracter <- Subcadena(expresion, i, 1)
    Sino
        // Fin de la cadena
    Fin Si
Fin Mientras
Si caracter = ")" Y Subcadena(expresion, i + 2, 1) = "{" Entonces
    esEstructuraDecision <- Verdadero
Fin Si
Fin Si

// Regla para espacios en blanco
Si caracter = " " O caracter = "\t" O caracter = "\n" Entonces
    esEspacioEnBlanco <- Verdadero
```



TECNOLÓGICO  
NACIONAL DE MÉXICO

## INSTITUTO TECNOLÓGICO DE CELAYA



### PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

#### AUTORES:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

*[Handwritten signatures]*

```
Fin Si

// Imprimir resultados
Si esIdentificador Entonces
    Escribir "IDENTIFICADOR: " + token
Sino Si esNumeroEntero Entonces
    Escribir "NÚMERO ENTERO: " + token
    Sino Si esDeclaracionVariable Entonces
        Escribir "DECLARACIÓN DE VARIABLE: " + token
        Sino Si esEstructuraDecision Entonces
            Escribir "ESTRUCTURA DE DECISIÓN: " + token
            Sino Si esEspacioEnBlanco Entonces
                // Ignorar espacios en blanco
                Fin Si
            Fin Si
        Fin Si
    Fin Si
Fin Si

// Restablecer banderas
esIdentificador <- Falso
esNumeroEntero <- Falso
esDeclaracionVariable <- Falso
esEstructuraDecision <- Falso
esEspacioEnBlanco <- Falso

Fin Mientras

FinAlgoritmo
```



# **PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II**

## **AUTORES:**

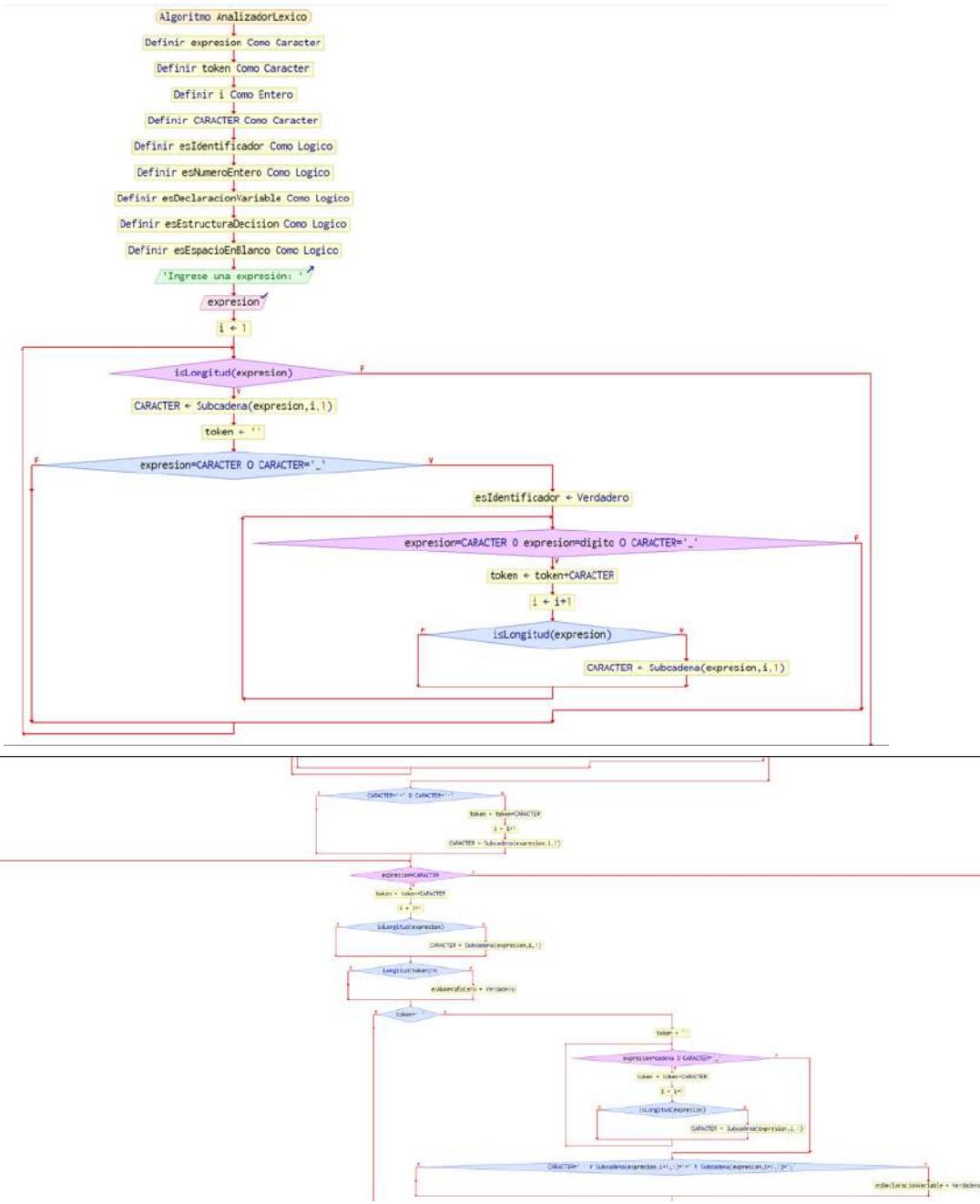
Alma Gabriela Ponce Morales

Anthony Gómez Cabañas

Cristhian Alberto Ortega Hernández

Miguel Ángel Ruiz López

## Diagrama de flujo





# **PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II**

## **AUTORES:**

Alma Gabriela Ponce Morales

Anthony Gómez Cabañas

Cristhian Alberto Ortega Hernández

Miguel Ángel Ruiz López

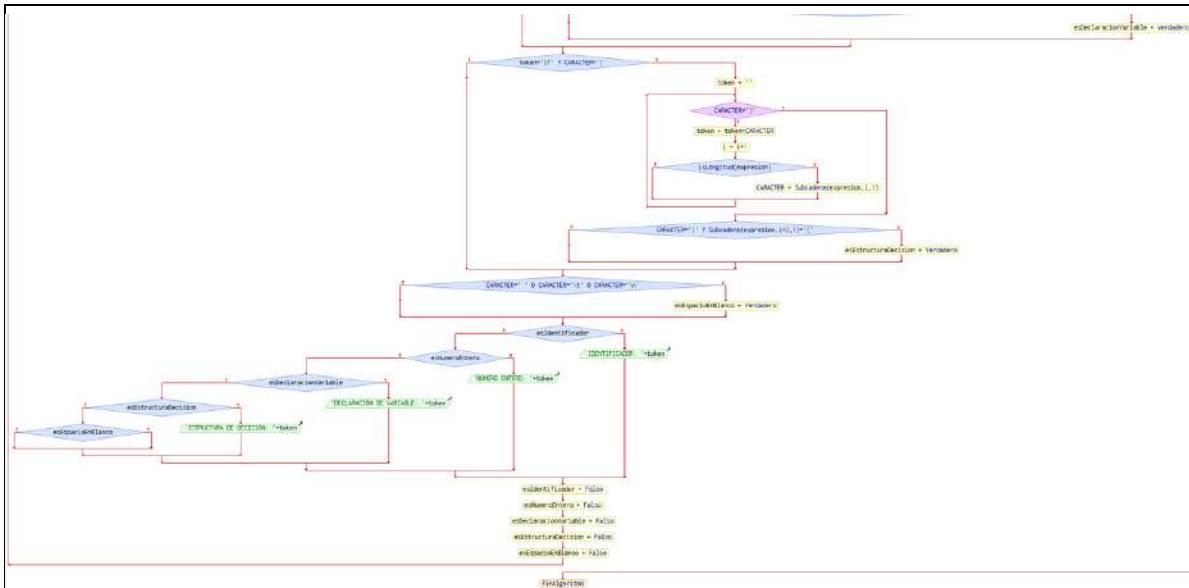


Imagen completa del diagrama de flujo:

<https://drive.google.com/file/d/1BqgQbKXagS7D12wP8JBHp92neABHnENu/view?usp=sharing>

## **Elaboración de la práctica**

Para esta práctica, como se mencionó en los requisitos, se hizo uso de un entorno que tuviera instalado una distribución de Linux. Por cuestiones de comodidad y porque ya se estaba trabajando de esta manera en otras materias, se utilizó WLS (Windows Subsystem for Linux), una capa de compatibilidad que permite ejecutar un entorno de Linux directamente en un sistema Windows sin la necesidad de una máquina virtual separada. En WSL se tiene instalado Ubuntu 22.04.

A continuación, se explica el desarrollo de la práctica:

Para comenzar, se abre una terminal en Windows PowerShell para poder ejecutar WSL. Una vez abierta esta terminal, utilizar el comando `wsl -d Ubuntu-22.04` para comenzar a ejecutar el sistema operativo.

```
PS C:\Users\crist> wsl -d Ubuntu-22.04
cristhian@CAOH-HP-Laptop:/mnt/c/Users/crist$ cd ~
cristhian@CAOH-HP-Laptop:~$
```

El sistema operativo inicia y se ejecuta el comando `cd ~` para ir al directorio de inicio. Ahora, se procede con la instalación de Flex, el generador de analizadores léxicos.

Para esto, primero se ejecutan los comandos `sudo apt update` y `sudo apt upgrade` para realizar un mantenimiento general al sistema operativo. Cuando haya terminado la actualización, se ejecuta el comando `sudo apt-get install flex` para instalar Flex.

TECNOLÓGICO  
NACIONAL DE MÉXICO**INSTITUTO TECNOLÓGICO DE CELAYA****PRÁCTICA DE LABORATORIO - LENGUAJES  
Y AUTÓMATAS II****AUTORES:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

```
alma@DESKTOP-SL3VS4E:~$ sudo apt-get install flex
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following packages were automatically installed and are no longer required:
  libaiol libmecab2 mecab-ipadic mecab-ipadic-utf8 mecab-utils mysql-community-client-plugins
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  binutils binutils-common binutils-x86_64-linux-gnu cpp cpp-12 gcc gcc-12 libabsl20220623 libasan8 libatomic1
  libavif15 libbinutils libc-dev-bin libc-devtools libcc1-0 libcrypt-dev libctf-nobfd0 libctf0 libfl-dev
  libfl2 libgavl1 libgcc-12-dev libgd3 libgomp1 libprofng0 libisl23 libitm1 liblsan8 libmpc3 libssl-dev libravie0
  libsvtavenc1 libtirpc-dev libtsan2 libubsan1 libx11-6 libx11-data libxau6 libxcb1 libxdmcp6 libxpm4 libyuv0
  linux-libc-dev m4 manpages manpages-dev rpcsvc-proto
Suggested packages:
  binutils-doc cpp-doc gcc-12-locales cpp-12-doc bison build-essential flex-doc gcc-multilib make autoconf automake
  libtool gdb gcc-doc gcc-12-multilib gcc-12-doc glibc-doc libgd-tools m4-doc man-browser
The following NEW packages will be installed:
  binutils binutils-common binutils-x86_64-linux-gnu cpp cpp-12 flex gcc gcc-12 libabsl20220623 libasan8 libatomic1
  libavif15 libbinutils libc-dev-bin libc-devtools libcc1-0 libcrypt-dev libctf-nobfd0 libctf0 libfl-dev
```

Ya con Flex instalado, se procede a la preparación de carpetas para la práctica. En este caso, se decidió crear una carpeta llamada “practicaslexyaac” para ahí guardar todas las prácticas que se realicen durante el semestre. Dentro de esa carpeta se creó otra llamada “practica1” para guardar los archivos que corresponden a esta práctica.

```
cristhian@CAOH-HP-Laptop:~$ mkdir practicaslexyaac
cristhian@CAOH-HP-Laptop:~$ ls
desarrolloweb practicaslexyaac
cristhian@CAOH-HP-Laptop:~$ cd practicaslexyaac
cristhian@CAOH-HP-Laptop:~/practicaslexyaac$ mkdir practical
cristhian@CAOH-HP-Laptop:~/practicaslexyaac$ cd practical
cristhian@CAOH-HP-Laptop:~/practicaslexyaac/practical$ cd ..
```

Ubicarse en la carpeta “practica1” para comenzar con la creación de archivos. Para crear los archivos, se puede hacer de dos maneras:

1. Desde la misma terminal, usando el editor de textos de su preferencia.
2. Mediante un editor de código.

Para esta práctica se decidió utilizar un editor de código ya que este proporciona mayores comodidades. El editor que se usó es Visual Studio Code, y para abrirlo sólo se ejecuta el comando `code ..`. Es importante mencionar que Visual Studio Code abre el directorio donde se está ubicado, por lo que, antes de ejecutarlo, se deberá ubicar en el directorio donde se desea crear archivos.

```
cristhian@CAOH-HP-Laptop:~/practicaslexyaac$ cd practical
cristhian@CAOH-HP-Laptop:~/practicaslexyaac/practical$ code ..
```

Con Visual Studio Code abierto, se procede a la creación de un archivo con extensión “.l”, esto indica que es un archivo que servirá como entrada para el generador de analizadores léxicos Flex. El archivo que se creó se llama “practica1.l”.



TECNOLÓGICO  
NACIONAL DE MÉXICO

## INSTITUTO TECNOLÓGICO DE CELAYA

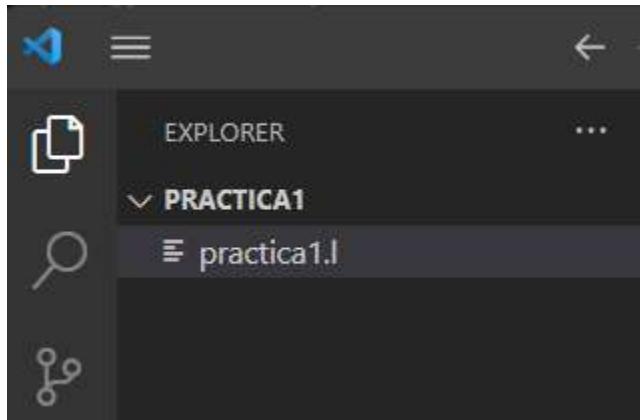


### PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

#### AUTORES:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

*[Handwritten signatures]*



En este archivo es donde se escribirán todas las expresiones regulares que permitirán al analizador léxico buscar sentencias válidas. A continuación, se muestra el código:

```
%{                                //Inicio de bloque para declaraciones de cabecera en C
    #include <stdio.h>           //Incluye la librería que permite entrada y salida
mediante teclado
%}                                //Fin de bloque

%%                                //Inicio de bloque para la declaración de tokens y
expresiones regulares
^[a-zA-Z_][a-zA-Z0-9_]*$          //Expresión regular de un identificador
{printf("IDENTIFICADOR\n");}        //Expresión regular de un identificador
[-+]?[0-9]+$                      //Número entero
ENTERO\n";}                         //Expresión regular de un número entero
[a-zA-Z]*[ ]+[a-zA-Z_][a-zA-Z0-9_]*[ ]*[ ]*[a-zA-Z0-9]+[;]$ //Expresión regular para la declaración
{printf("DECLARACIÓN DE VARIABLE\n");} //Expresión regular para la declaración
de una variable
if[ ]*\([ ]*\)[ ]*\{[^}\]*\}$      //Expresión regular de un if
DE DECISIÓN\n");}                  //Expresión regular de un if
[ \t\n]                             ;
//Espacio en blanco
.
DESCONOCIDO\n");}                  //Carácter desconocido
%$                                //Fin de bloque
```

El código comienza con la cabecera, en este bloque se incluyen librerías o archivos necesarios. Para este ejemplo, se incluyó la librería que permite ingresar datos por teclado. Posteriormente, está el bloque donde se declaran las expresiones regulares. A continuación, se explican estas expresiones:

1. `^[a-zA-Z_][a-zA-Z0-9_]*$`: Esta es una expresión regular para un identificador. Debe de



# **PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II**

## **AUTORES:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

iniciar con una letra minúscula, mayúscula o guión bajo, después pueden seguir letras en minúscula, mayúscula, números o guión bajo. Después de eso ya no detectará otro carácter.

2. `^[a-zA-Z_][a-zA-Z0-9_]*$`: Es una expresión regular para un número entero. De manera opcional, pide se inicie con signos, después, un número de 0 a 9 que se puede repetir.
  3. `[a-z]*[ ]+[a-zA-Z_][a-zA-Z0-9_]*[ ]*[=][ ]*[a-zA-Z0-9]+[ ;]$`: Expresión regular para la definición de una variable. Inicia con el tipo de dato, nombre de la variable, =, valor de la variable, valor. Así se configuran todas las entradas.
  4. `if[ ]*\{[^ ]*\}[ ]*\{[^ ]*\}\}$`: Identifica cuando se declara un if. Para que lo detecte, se debe de iniciar con la palabra if, después un paréntesis de apertura y, adentro, una condición. Después se coloca un paréntesis de cierre y se abre una llave, ahí adentro se coloca alguna otra instrucción y se cierra la llave.

Las siguientes dos expresiones son simplemente para que se ignoren espacios en blanco y cualquier otro carácter que no sea correcto.

Ese fue el programa que se elaboró para la práctica, ahora queda ejecutarlo y ponerlo a prueba. Para esto, en la terminal se ejecutó el comando `flex practical1.l`, que generó un archivo “`lex.yy.c`” que contiene código fuente en C para poder crear un ejecutable. Después se ejecutó el comando `gcc -o practical1 lex.yy.c -lfl` para finalmente crear el ejecutable.

```
cristhian@CAOH-HP-Laptop:~/practicaslexyaac/practical$ flex practical.l  
cristhian@CAOH-HP-Laptop:~/practicaslexyaac/practical$ ls  
lex.yy.c  practical.l
```

```
cristhian@CAOH-HP-Laptop:~/practicaslexyaac/practical$ gcc -o practical lex.yy.c -lfl
cristhian@CAOH-HP-Laptop:~/practicaslexyaac/practical$ ls
lex.yy.c  practical  practical.l
```

Para ejecutar el programa, se ingresa el siguiente comando ./practica1.

## Resultados

TECNOLÓGICO  
NACIONAL DE MÉXICO**INSTITUTO TECNOLÓGICO DE CELAYA****PRÁCTICA DE LABORATORIO - LENGUAJES  
Y AUTÓMATAS II****AUTORES:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

*[Handwritten signatures]*

```
1
NÚMERO ENTERO
+143
NÚMERO ENTERO
-987
NÚMERO ENTERO
123p
CARACTER DESCONOCIDO
CARACTER DESCONOCIDO
CARACTER DESCONOCIDO
CARACTER DESCONOCIDO
```

```
int mi_variable = 5;
DECLARACIÓN DE VARIABLE
char caracter = c;
DECLARACIÓN DE VARIABLE
if (a<b) {a=a+1;}
ESTRUCTURA DE DECISIÓN
```

Con esto concluye el desarrollo de la práctica.

BÍTACORA DE INCIDENCIAS		
Fecha	Problema encontrado	Solución
23/09/2023	Flex no funcionó en el WSL de un integrante.	Se instaló una máquina virtual en VirtualBox.
24/04/2023	La expresión regular para los identificadores permitía entradas que no deberían de ser válidas.	Se revisaron algunos ejemplos en internet y se encontraron operadores que hacían falta en la expresión.
24/04/2023	La expresión regular para los números permitía entradas que no deberían ser válidas.	Con la solución que se encontró para el problema anterior se pudo solucionar este.

CONCLUSIÓN	
El desarrollo de esta práctica fue de mucha ayuda para comprender de mejor manera la importancia del análisis léxico y los analizadores léxicos para el desarrollo de compiladores. Aprendimos a utilizar la herramienta Flex y conocimos la ventaja más importante que esta proporciona, generar analizadores léxicos.	
Entre todo el equipo, consideramos que la práctica fue de gran importancia para comprender de mejor	



TECNOLÓGICO  
NACIONAL DE MÉXICO

## INSTITUTO TECNOLÓGICO DE CELAYA



### PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

#### AUTORES:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

manera todo lo que se investigó para la parte teórica de esta actividad.

8

#### REFERENCIAS

- Compilador Diseño - Análisis Léxico.* Online Courses and eBooks Library. (n.d.).  
[https://www.tutorialspoint.com/es/compiler\\_design/compiler\\_design\\_lexical\\_analysis.htm](https://www.tutorialspoint.com/es/compiler_design/compiler_design_lexical_analysis.htm)
- Cartagena99. (n.d.).  
[https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2\\_M4\\_U2\\_T1.pdf](https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2_M4_U2_T1.pdf)
- Fase de Análisis Léxico del Compilador.* RicardoGeek. (2017, January 10).  
<https://ricardogeek.com/fase-de-analisis-lexico-del-compilador/>
- MozDevNet. (n.d.). *Expresiones Regulares - JavaScript: MDN.* JavaScript | MDN.  
[https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular\\_expressions](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_expressions)
- Ricardocelis. (2018, June 20). *¿Qué son las expresiones regulares? Explicadas en video.* Platzi. <https://platzi.com/blog/que-expresion-regular/>
- Analizador Léxico.* EcuRed. (n.d.). [https://www.ecured.cu/Analizador\\_Léxico](https://www.ecured.cu/Analizador_Léxico)
- Analizador Lexico - analizadores Léxicos. act. 6: En lenguaje de programación.* Instituto Tecnológico. Studocu. (n.d.). <https://www.studocu.com/es-mx/document/instituto-tecnologico-superior-de-irapuato/taller-de-la-investigacion/analizador-lexico/31255772>
- Lex: Un generador de analizadores Léxicos - unizar.es. (n.d.-b).  
[https://webdiis.unizar.es/~ezpeleta/lib/exe/fetch.php?media=misdatos:compi:2bis.intro\\_ex.pdf](https://webdiis.unizar.es/~ezpeleta/lib/exe/fetch.php?media=misdatos:compi:2bis.intro_ex.pdf)
- 4 ingenier a inform Atica - Uji. (n.d.-a).  
<https://repositori.uji.es/xmlui/bitstream/handle/10234/5877/lexico.apun.pdf?sequence=1>
- Lucas, J. (2023, April 17). *Qué es C: Características Y Sintaxis.* OpenWebinars.net.  
<https://openwebinars.net/blog/que-es-c/>



TECNOLÓGICO  
NACIONAL DE MÉXICO

## INSTITUTO TECNOLÓGICO DE CELAYA



### PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

#### AUTORES:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

*[Handwritten signatures]*

Saavedra, J. A. (2023, June 1). *Qué es lenguaje c: El Origen, Las Ventajas, Las características y la Sintaxis del Lenguaje de Programación*. Ebac.

<https://ebac.mx/blog/que-es-lenguaje-c>

Elementos Léxicos del Lenguaje de programación C - universidad de granada. (n.d.).

<https://elvex.ugr.es/decsai/c/apuntes/tokens.pdf>

Gangadhar S. (2018, 7 marzo). *How to install LEX & YACC packages into Ubuntu // How to*

*run LEX & YACC programs in Ubuntu with step*. YouTube. Recuperado 26 de septiembre de 2023, de <https://www.youtube.com/watch?v=FyR3fx4qiZQ>

Jonathan Engelsma. (2013, 14 febrero). *Part 01: Tutorial on Lex/YACC* [Vídeo]. YouTube.

<https://www.youtube.com/watch?v=54bo1qaHAfk>

IBM Developer. (s. f.). <https://developer.ibm.com/tutorials/au-lexyacc/>

*[Handwritten signatures]*

# INSTITUTO TECNOLÓGICO DE CELAYA

Lenguajes y Autómatas II

Equipo 2

Gomez Cabañas Anthony 20030057

Ortega Hernández Cristhian Alberto 20031091

Ponce Morales Alma Gabriela 20030274

Ruiz López Miguel Ángel 20030935

Monografía 2.  
Gramáticas Libres de  
Contexto y su  
Representación Visual

I.S.C. Ricardo González González

# ÍNDICE

Índice de recursos	2
Introducción	3
Generalidades	5
Desarrollo	7
Gramáticas Libres de contexto y árboles	
de Derivación	
1 <b>EI</b> analizador sintáctico o parser	7
2 <b>GCLC</b> o gramática libre de contexto	10
2.1 Definición de una GCLC	12
3 Arboles de Derivación	15
3.1 Derivaciones	15
3.2 Derivaciones Izquierda y Derecha	16
3.3 Árboles de derivación o parseo	17
3.4 Gramática Ambigua	22
4 Diagramas de Sintaxis	23
4.1 Estructura de los diagramas de sintaxis	24
4.2 Ejemplos de diagramas de sintaxis	27
Conclusiones	34
Bibliografía	36

# ÍNDICE DE RECURSOS

Ilustración 1: Se muestra la interacción de un analizador sintáctico con el resto de las fases del compilador. Imagen de Teoría, Diseño e implementación de compiladores de lenguajes 9

Ilustración 2: Estructura de un árbol de derivación 18

Ilustración 3: Árbol de derivación de derivación para P<sub>0110</sub> 19

Ilustración 4: Árbol de derivación a la izquierda para aabaa 20

Ilustración 5: Árbol de derivación a la derecha para aabaa 20

Ilustración 6: Elemento necesario 24

Ilustración 7: Elemento repetible 24

Ilustración 8: Elemento opcional 24

Ilustración 9: Apertura y cierre 25

Ilustración 10: Componente léxico 26

Ilustración 11: Símbolo especial 26

Ilustración 12: Símbolo no terminal 26

Ilustración 13: Diagrama de secuencia De Sentencias 27

Ilustración 14: Diagrama de sentencia 28

Ilustración 15: Diagrama de sentencia Case 29

Ilustración 16: Diagrama del terminal artículo 29

Ilustración 17: Diagrama del terminal sustantivo 30

Ilustración 18: Diagrama del terminal verbo 30

Ilustración 19: Diagrama del NO terminal predicado 30

Ilustración 20: Diagrama del NO terminal sujeto 30

Ilustración 21: Diagrama del NO terminal frase 31

# Introducción

La gramática es un elemento fundamental que sirve como herramienta para comprender la estructura de los lenguajes, ya sean naturales o artificiales/formales. La importancia de las gramáticas libres de contexto (GLC) en este ámbito es primordial al permitir éstas la descripción de patrones gramaticales en una amplia gama de lenguajes. El presente trabajo se busca adentrar en el estudio de las GLC y explorar su aplicación en la generación y análisis de lenguajes formales.

La capacidad de las GLC para representar estructuras sintácticas y generar lenguajes de manera eficiente las convierte en un recurso invaluable en la resolución de problemas complejos relacionados con la comunicación y la automatización del lenguaje.

En este contexto, el problema que se plantea se relacion con la necesidad de comprender en profundidad las gramáticas libres de contexto, así como su aplicación práctica en la construcción de árboles de derivación y diagramas de sintaxis. Además, se busca delimitar este problema enfocándose específicamente en el análisis de la

relación entre las GLC y la representación visual de la estructura gramatical a través de diagramas de sintaxis. Esta delimitación permitirá explorar con detalle el funcionamiento y la utilidad de estas herramientas en el contexto de la teoría de lenguajes formales.

A medida que avance este trabajo, se examinará en profundidad los conceptos fundamentales de las GLC, la construcción de árboles de derivación como herramientas de análisis y, finalmente, la representación gráfica de la estructura gramatical mediante diagramas de sintaxis.

# Generalidades

Comprender las Gramáticas Libres de Contexto es imprescindible en muchos campos de la computación como la inteligencia artificial, compiladores o procesamiento del lenguaje natural. Entender cómo funcionan brinda herramientas poderosas para describir patrones gramaticales en lenguajes.

Así, para poder estudiarlas, es de gran ayuda hacer uso de representaciones visuales, pues así se logra una mayor comprensión de estructuras gramaticales complejas. Herramientas como los árboles de derivación o los diagramas de sintaxis actúan como puentes entre la teoría lingüística y la visualización sintáctica de las gramáticas, proporcionando una forma de entendimiento más sencilla, rápida y digerible para el lector al tratarse de representaciones gráficas de las gramáticas subyacentes en lenguajes formales y naturales.

En la presente monografía, se pretende explorar a fondo las GLC, desglosando y entendiendo su funcionamiento y mostrar las aplicaciones que pueden tener. Además, se explicarán los elementos gráficos que éstas emplean, como los árboles de derivación, para representar la secuencia precisa

de derivaciones que transforman una cadena de símbolos en una expresión gramaticalmente correcta; y los diagramas de sintaxis, que proporcionan una vista panorámica de la estructura gramatical de un lenguaje y su aplicabilidad en el procesamiento del lenguaje natural, la programación y otros campos.

Todos estos temas desempeñan un papel esencial en la teoría de lenguajes formales y tienen aplicaciones significativas en la resolución de problemas técnicos y computacionales. A través de ejemplos y análisis detallados, se demostrará cómo estas herramientas proporcionan una comprensión sólida y accesible de la estructura gramatical en lenguajes formales y naturales, lo que las convierte en recursos valiosos en un mundo impulsado por la comunicación digital y la automatización.

# Desarrollo

## Gramáticas Libres de Contexto y Árboles de Derivación

### I El analizador sintáctico o parser

Como forma de introducción y para comprender la tarea que deben realizar los compiladores y interpretes recordamos que ambas son similares ya que ambos se encargan de convertir el código escrito a un formato legible para la máquina (mismo que conocemos como lenguaje máquina) pero cuentan con diferencias en cuanto a la manera en la que realizan dicha conversión.

Según un artículo en línea publicado por el proveedor de hosting web IONOS, dicha diferencia radica principalmente en que el intérprete realiza la conversión al momento de ejecutar el software y únicamente sobre las líneas modificadas desde la última ejecución. Mientras que el compilador realiza dicha antes de ejecutar el programa, además que siempre será sobre todo el código sin importar cuán mínimo (o inexistente) sea el cambio.

Dicho lo anterior en este escrito se enfocara al ámbito de los compiladores, que como mencionan los autores del libro "Teoría, diseño e implementación de Compiladores de Lenguajes," esta misión o tarea se divide en dos superfases:

*"Primera superficie: compilar el código fuente original, avisando al programador de los errores de implementación que este haya cometido.*

*Segunda superficie: generar el código intermedio y adecuado para ser ejecutado"*

*(Martínez López & Romalvo, 2015).*

Dentro de la primera superficie mencionada anteriormente tienen lugar tres tipos de análisis los cuales son:

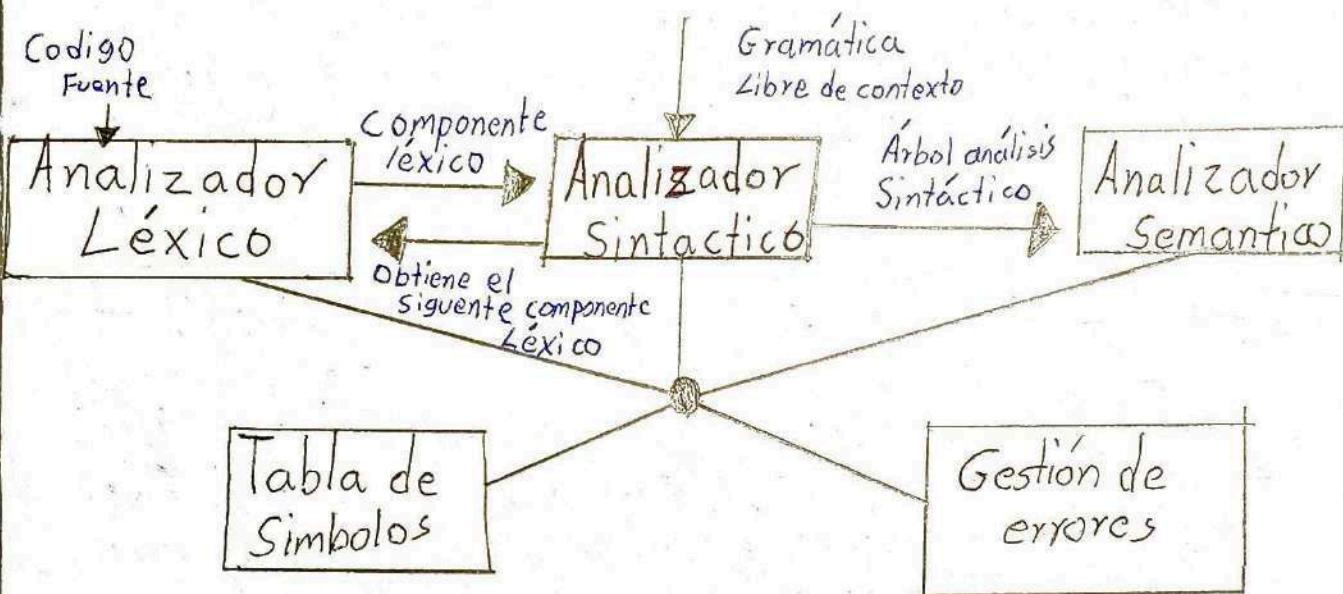
léxico, sintáctico y semántico. El sintáctico es la que más nos compete y será la que se abordara, y es que muchos autores como Alfonseca Moreno consideran esta como la fase primordial o motora de un compilador debido a su alta importancia para este mismo.

Una gramática libre de contexto o por sus siglas GLC (también se conoce como independiente del contexto) será la encargada de definir la fase de análisis léxico como la de análisis sintáctico, se menciona dicha ya que se hará mención en este capítulo (ya que esta altamente ligada con esta etapa) y en el siguiente se proporcionará una definición más formal sobre esta misma.

Así pues las tareas que desempeña dicho analizador sintáctico son:

- Obtener una cadena de tokens (proporcionados por el análisis léxico) y a partir de esta determinar si dicha cadena puede o no ser generada por la gramática fuente, es decir del lenguaje fuente (la gramática será libre de contexto).

- Apartir de lo anterior, generara el árbol sintáctico que definirá la jerarquía de un programa y obtendrá la serie de derivaciones para generar la cadena de componentes léxicos.
- Además si se presenta un error sintáctico este deberá hacer llegar al programador dichos de forma clara, precisa y significativa. Así tambien deberá ser capaz de contar con un mecanismo de recuperación de errores para que pueda continuar con el procesamiento del resto de la entrada.



**Ilustración 1:** Se muestra la interacción de un analizador sintáctico con el resto de las fases del compilador. Imagen extraída de "Teoría, Diseño e implementación de compiladores de lenguajes".

En la ilustración 1 observamos a detalle la interacción que se da entre el analizador sintáctico y todos los demás procesos que se dan en los tres tipos de análisis.

SPP  
A.M.C.

Para complementar el punto numero tres de las tareas que deberá cumplir el analizador sintáctico es decir aquel referente a los errores sintácticos. La información que se hará llegar al usuario es aquella que ya se habrá visto en alguna otra ocasión como lo sería el mensaje de "syntax error" o también decir el lugar concreto donde se encuentra dicho error en el programa fuente, por ejemplo "error en la línea 15", también se podría incluir un apuntador o señal que indique en qué carácter inicia dicho error.

Ademas varios compiladores incorporan sugerencias para corregir el error encontrado, como por ejemplo: "Falta un punto y coma al final de la sentencia de la linea 15".

## — II GLC o Gramática Libre de Contexto —

En este capítulo se profundizara sobre las GLC, estas mismas son las responsables de establecer la sintaxis y estructura de un lenguaje de programación, ya que como se había mencionado anteriormente esta gramática definirá tanto la fase de análisis sintáctico como el de análisis léxico ya que es usado para especificar la estructura de los diferentes tokens que regresara este último.

Recordando la jerarquía de Chomsky para gramáticas generadoras de lenguajes formales, una GLC es definida como una gramática de tipo dos, y haciendo mención que los componentes léxicos que se reciben por parte de/

Aula 10

analizador léxico son definidos por expresiones regulares  
(es decir una gramática tipo tres) es importante mencionar que:

"Un lenguaje regular (producido por una gramática tipo tres) es una especialización del lenguaje libre de contexto y, por lo tanto toda expresión regular se puede construir con producciones de una GLC" (Martínez López & Ramallo, 2015).

Por lo tanto una GLC usa convenciones de nombramiento y operaciones de forma similar a las usadas en las expresiones regulares, pero presenta una diferencia la cual dice que se utiliza la GLC en lugar de las expresiones regulares y esta es que **las reglas de una GLC son recursivas.**

A partir de lo anterior podemos plantear un ejemplo relacionado al bucle o sentencia while, veamos que es parte de su estructura el permitir que otros bucles while puedan estar anidados a este mismo. Esto es posible gracias a la recursividad de las GLC antes mencionadas y si quisieramos realizar esto mismo con una expresión regular simplemente no sería posible.

Entonces aparentemente dicho cambio parece muy situacional pero la realidad es que tiene un peso considerable. Ya que debido a la recursividad las estructuras o cadenas reconocidas por las GLC aumentan considerablemente.

Aula 11  
2023

De forma resumida las mayores restricciones al usar expresiones regulares para definir un lenguaje, es el no poder utilizar construcciones anidadas (mismas que son base fundamental en los lenguajes de programación) como lo vendría siendo parentesis equilibrados (por cada parentesis abierto, se debe contar ademas con su cierre o final), pares de palabras clave como begin-end, do-while, <html>-</html>, bucles o condicionales anidados.

Con todo lo expuesto anteriormente, podemos concluir que simplemente no es viables estructurar un lenguaje de programación únicamente con expresiones regulares ya que estaría considerablemente mermado y su expresividad sería muy limitada.

## II. I Definición de una GLC

La definición de una GLC es regularmente la misma a través de los autores, o libros donde se consulte, pudiesen existir variaciones muy ligeras, pero éstas son prácticamente las mismas debido a que al utilizar una GLC se está utilizando un metalenguaje para expresar dicha gramática, este metalenguaje recibe el nombre de notación BNF (Forma de Backus-Naur).

Una vez dicho lo anterior, una GLC se define por la cuádrupla siguiente:

$$G = \{V, T, P, S\}$$

- APELLIDOS
- \*  $V$  es el conjunto finito de símbolos o variables no terminales (o a veces denominados categorías sintácticas), donde cada una representa un conjunto de cadenas (es decir un lenguaje).
  - \*  $T$  es el conjunto finito de símbolos o variables terminales de la gramática. Suelen ser los distintos tokens o componentes léxicos del lenguaje.
  - \*  $P$  es el conjunto finito de reglas o producciones que representan la definición recursiva de un lenguaje, donde cada producción constará de:
    - Una cabeza de la producción que es una variable que define parcialmente la producción
    - El símbolo de producción " $\rightarrow$ "
    - Cuerpo de la producción que es una cadena formada por cero (cadena vacía) o mas símbolos terminales y variables. Esta indicara la forma o estructura de las cadenas que pertenezcan al lenguaje de la variable de la cabeza. Mediante este lineamiento se dejan los símbolos terminales sin modificación y se pueden intercambiar cada variable del cuerpo por una cadena que pertenezca al lenguaje de la variable.
  - \*  $S$  sera la variable que representará el lenguaje que se estará definiendo, se conoce como Símbolo inicial.

*Agradecido*

El convenio o pautas de notación para representar los símbolos/componentes anteriores en gran medida es la misma y se recomienda encarecidamente su uso ya que mediante esta, se genera un estándar que cualquier persona con conocimientos de este comprenderá evitando barreras o limitantes al revisar el trabajo de terceros.

Así pues en los símbolos terminales o T se recomienda:

- Uso de letras minúsculas del comienzo del alfabeto
- Operadores como +, -, \*, /
- Símbolos de puntuación como lo serían parentesis o coma etc.
- Dígitos del 0 - 9.
- Cadenas subrayadas o en negrita como id, if
- Nombres en mayúsculas como Begin, end (BEGIN, END).

En símbolos no terminales o V se recomienda:

- Uso de las letras mayúsculas del comienzo del alfabeto como A, B, C, D.
- Uso de la letra S para representar el símbolo inicial
- Nombres en minúsculas como expr o sente,

En producciones o P se recomienda:

- El uso de las letras griegas minúsculas como podría ser " $\alpha$ ,  $\beta$  o  $\gamma$ " ya que estas representan cadenas de símbolos gramaticales (tanto terminales como no). A partir de esto se puede escribir una

Aula Cálculo

generica de la forma:  $A \rightarrow d$

Si tenemos  $A \rightarrow d_1, A \rightarrow d_2, A \rightarrow d_3, \dots$  y así sucesivamente entonces  $A \rightarrow d_m$  (o a veces  $\kappa$ ) serán todas las producciones con el mismo lado izquierdo de la producción. A veces reciben el nombre de alternativas de  $A$  y se pueden representar de la forma:

$$A \rightarrow d_1 | d_2 | d_3 | \dots | d_m.$$

## — III Árboles de Derivación —

### III. I Derivaciones

Antes de aterrizar formalmente el concepto de los árboles de derivación primero es necesario plantear el concepto de derivaciones en el contexto de una GLC.

Así pues si aplicamos las producciones propias de una GLC para realizar una inferencia que nos servirá para determinar que cadenas pertenecen al lenguaje de una variable determinada, tendremos dos maneras o métodos para llevar a cabo dicha inferencia. La más habitual consiste en tomar cadenas ya conocidas que pertenecen al lenguaje de cada variable del cuerpo y estas mismas las concatenamos en el orden adecuado con cualquier símbolo terminal, mismo que aparezca en el cuerpo a partir de esto podemos inferir qué la nueva

Aprobado  
Año 2011

cadena resultante deberá por lo tanto pertenecer al lenguaje de la variable del encabezado. Este procedimiento recibe el nombre de inferencia recursiva.

El segundo método consiste en expandir el símbolo inicial usando una de sus producciones (es decir una en donde la cabeza de dicha sea el símbolo inicial).

Para posteriormente expandir la cadena resultante reemplazando una variable elegida por el cuerpo de alguna de sus producciones, y así sucesivamente, esto hasta que se obtenga una cadena compuesta enteramente por terminales. Así entonces el lenguaje de dicha gramática Serán todas las cadenas que se puedan obtener por dicha forma. Este método tiene el nombre de derivaciones y es el principio básico de los árboles de derivación o de parseo.

La notación o convenio para dichas derivaciones serán prácticamente las mismas establecidas en el capítulo anterior ya que como su nombre indica, son derivados del mismo entonces tiene sentido que mantengan la misma estructura.

### III. II Derivaciones Izquierda y Derecha

Las maneras en las que se puede realizar dicho reemplazo mencionado anteriormente son dos y están planteadas de manera que se restrinja la cantidad de opciones disponibles para dicha derivación.

Aprobado

de una cadena en concreto son las siguientes:

\* Derivación a la izquierda o mas a la izquierda:  
en esta se reemplaza como su nombre dice, la  
variable que se encuentre mas a la izquierda por  
algún cuerpo de sus producciones. La manera de  
indicarlo es mediante la relación:  $\xrightarrow{im}$  y  $\xrightleftharpoons[im]{*}$

\* Derivación a la derecha o mas a la derecha:  
Mismo procedimiento que la anterior, pero  
ahora se reemplaza la variable que se encuentre  
mas a la derecha por algún cuerpo de sus produc-  
ciones. La manera de indicarlo es mediante la  
relación:  $\xrightarrow{rm}$  y  $\xrightleftharpoons[rm]{*}$

### III. Arboles de derivación o parseo

Entonces ya con los conceptos anteriores se puede llegar a la pregunta siguiente ¿Qué es un arbol de derivación? Pues sencillamente es una representación gráfica de como se puede derivar cualquier cadena de un lenguaje a partir de un símbolo distinguido, como su nombre indica es una estructura de arbol, en otras palabras es un grafo conexo y que tiene un único modo raíz, este además no contara con ciclos esde-cir si queremos llegar a un nodo C desde un nodo A existirá un único camino para llegar a dicho nodo.

Con dichos árboles resulta más sencillo y de una manera visual el identificar o producir derivaciones pero el factor más importante radica en que cuando este es utilizado en un compilador, este árbol es la estructura de datos que representa el programa fuente. Esto facilita la traducción del programa fuente a código máquina ejecutable ya que dicho proceso será realizado por funciones naturales recursivas.

Su definición es la siguiente: Dada una GLC =  $EV, T, P, S,$  es decir una gramática libre de contexto se tiene:

- \* **Nodo raíz:** Es el símbolo inicial ( $S^*$ )
- \* **Nodos interiores:** Corresponden a los símbolos no terminales ( $V$ )
- \* **Nodos hoja:** Corresponden a los símbolos terminales ( $T$ ) o también la cadena vacía  $\epsilon$  solo si es hijo único.

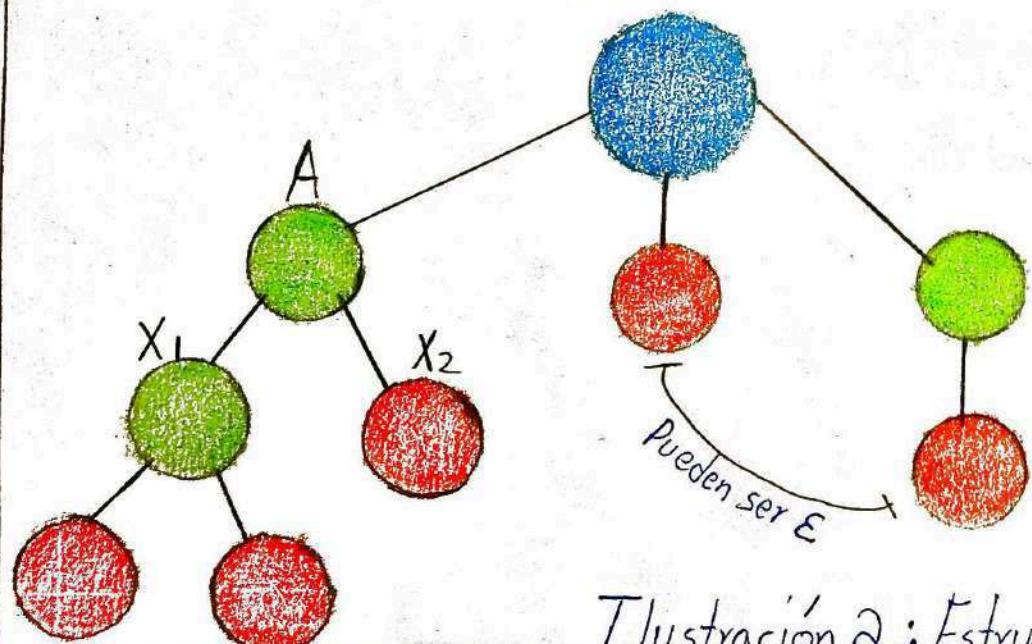


Ilustración 2: Estructura de un Árbol de derivación

\* Finalmente las reglas de derivación se representan por los nodos interiores y por sus hijos.

A partir de la ilustración anterior de A se pueden generar la cadena  $A \rightarrow X_1 X_2 \dots X_k$  donde  $X_2$  es un símbolo terminal pero  $X_2$  es un no terminal que se necesitará seguir derivando.

Entonces se tomara un ejemplo presentado por el profesor Dr. Fabián Riquelme Csori en un video en youtube de nombre "Lenguajes y Autómatas - Módulo 2.2 Árboles de derivación"

Dicho ejemplo consiste en una gramática generadora de palíndromos.

$$P \rightarrow \epsilon \quad P \rightarrow O \quad P \rightarrow I \quad P \rightarrow OPO \quad P \rightarrow 1P1$$

Y se construye un árbol de derivación para la cadena  $P \Rightarrow^* 0110$

Quedando de la siguiente manera:

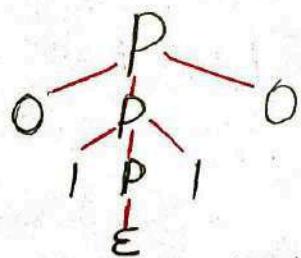


Ilustración 3: Árbol de derivación para  $P \Rightarrow 0110$

~~Aula C~~

Y retomando el concepto de derivación a la más izquierda y derivación mas a la derecha, tambien se cuenta con su equivalente aplicado a los árboles de derivación donde se tienen los siguientes tipos de árboles:

- \* Árboles de derivación a la izquierda: Obteniéndolo al aplicar la producción a la variable más a la izquierda de cada paso.
- \* Árbol de derivación a la derecha: Obtenido al aplicar la producción a la variable más a la derecha de cada paso.

Su uso es el mismo que el definido anteriormente y es para limitar la cantidad de opciones disponibles para dicha derivación de una cadena.

A continuación se desarrolla un ejemplo que mostrara la misma derivación pero en el primero se realizará una derivación a la izquierda y en el segundo una derivación a la derecha para ver sus diferencias.

El enunciado es: Construir el arbol de derivación para la cadena aabaa de la gramática

$$S \rightarrow aAS \mid aS \mid \epsilon, A \rightarrow SbA \mid ba$$

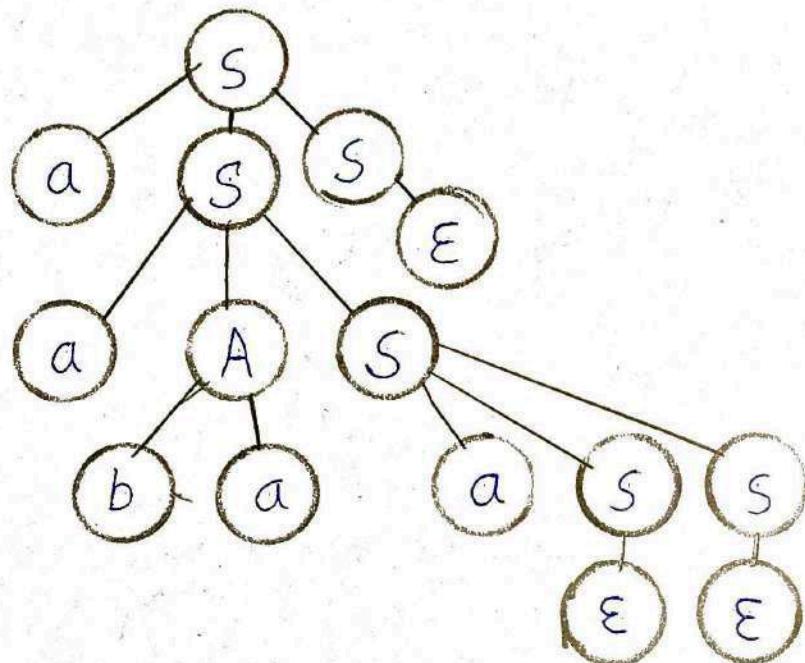


Ilustración 4: Árboles de derivación a la izquierda  
para aabaa

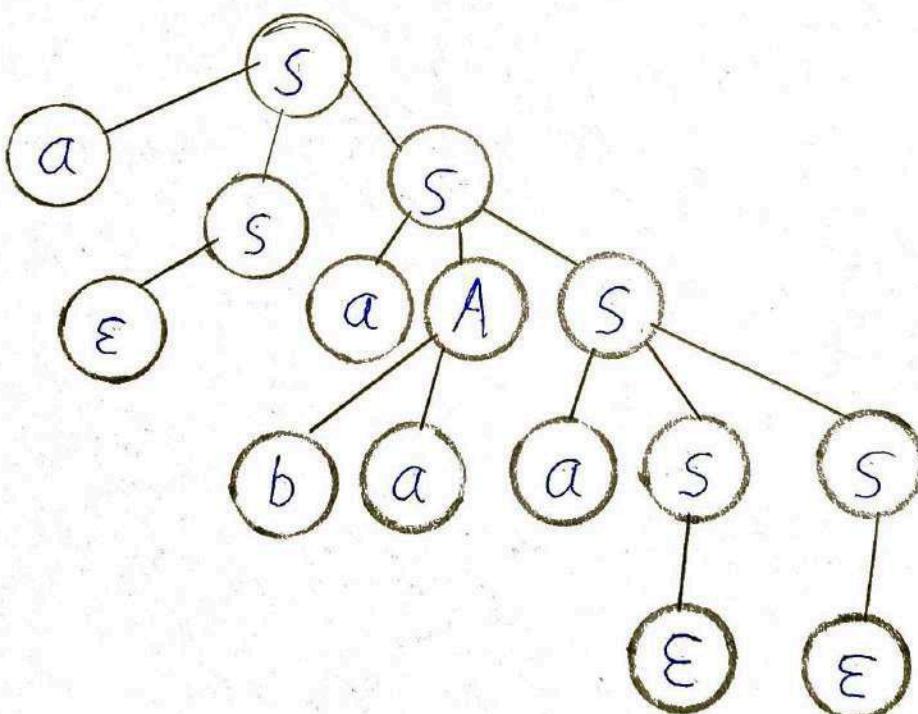


Ilustración 5: Árbol de derivación a la derecha  
para aabaa

### III. IV Gramática Ambigua

Se tiene la suposición que una gramática siempre generara una estructura única para cada cadena del lenguaje, pero esta suposición no siempre es correcta y existieran ocasiones en las que no todas las gramáticas devolverán estructuras únicas.

Cuando dicho comportamiento se presente existen ocasiones en las que se pueda rediseñar para eliminar la ambigüedad pero existirán en ocasiones GLC que serán inherentemente ambiguas es decir que no se puede eliminar dicha ambigüedad.

Se dice que cuando una gramática produce mas de un árbol de derivación para la misma sentencia entonces esta es una gramática ambigua. La implicación radica en que dicha ambigüedad significa que una misma sentencia se le pueden asignar significados (semánticas) diferentes.

Resulta contraproducente que en los lenguajes formales (lenguajes de programación, lógica, matemáticas) exista ambigüedad ya que esto implica que existan diferentes significados para por ejemplo un mismo programa (resultando en que el compilador devuelva diferentes códigos máquina) o en una fórmula matemática podamos llegar a resultados diferentes para algo que en teoría debe ser fijo e único.

# Diagramas de Sintaxis

Los diagramas de sintaxis más antiguos que respectan al análisis de los lenguajes, tanto formales como naturales, en informática, estaban constituidos de manera gráfica por mecanismos que, mediante refinamientos sucesivos, se determinaba si un lenguaje era admisible por una gramática o no.

En otras palabras, estos diagramas son formas gráficas de representar el nivel sintáctico de los lenguajes de programación, aunque no sólo se limitan a éstos (más adelante se mostrará un ejemplo de su empleo con lenguajes naturales).

Un diagrama de sintaxis, también conocidos como diagramas de Conway, puede valerse de otros diagramas de sintaxis para representar gráficamente la estructura de la gramática.

De hecho, lo más normal es que se utilicen múltiples diagramas para explicar detalladamente

*[Handwritten signatures]*

el comportamiento de la sintaxis para representar bien las gramáticas.

## Estructura de los diagramas de Sintaxis

Hoy en día, la estructura y propósito se de éstos no ha cambiado mucho. A grandes rasgos, un diagrama de sintaxis es un grafo dirigido que utiliza símbolos para representar elementos de una instrucción. Estos símbolos se clasifican en dos: Terminales (Constantes) y No Terminales (Variables); y se encuentran dentro de nodos. Estos diagramas también arcos que expresan las secuencias en que pueden combinarse los símbolos para formar frases aceptables según la gramática.

Cada diagrama de sintaxis representa una variable o símbolo no terminal (que se puede expandir), de manera que la gramática completa está formada por distintos símbolos interrelacionados

Como no terminales que se quieran incluir en su descripción. Los símbolos terminales o constantes, también conocidas como tokens o palabras, se dibujan como círculos o elipses, mientras que los no terminales se representan en un grafo rectangular etiquetado con su nombre correspondiente.

Los elementos necesarios aparecen en la línea horizontal o línea de acceso principal

►—————necesario————►

Ilustración 6. Elemento Necesario

Los elementos opcionales aparecen bajo la vía de acceso principal.

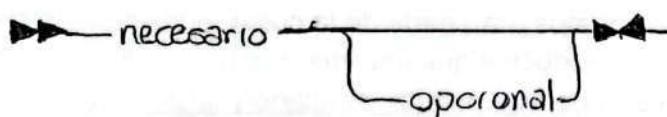


Ilustración 7. Elemento Opcional

Una flecha que vuelve a la izquierda sobre la línea principal, indica que un elemento se puede repetir.

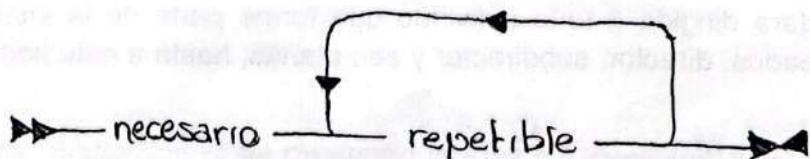


Ilustración 8. Elemento Repetible

En algunas notaciones, como la que utiliza IBM, las comillas angulares de cierre y guión '»-' ó '>>-' indican el principio de un diagrama de sintaxis.

Los guiones seguidos, por su parte, de una comilla angular de cierre y una comilla angular de apertura '-->' ó '→' indican el final de un diagrama de sintaxis.



Ilustración 9. Apertura y Cierre

De manera convencional, los diagramas de sintaxis se leen de izquierda a derecha y de arriba a abajo, siguiendo la vía de acceso de la línea. Es por esto que se asume que todo diagrama de sintaxis pasee un origen a la izquierda y un destino a la derecha.

## Notación

- Elipse/Rectángulo Ovalado: Representa el valor de un componente léxico, análogo a símbolos terminales de las gramáticas libres de contexto (GLC).

*[Handwritten signatures]*



### Ilustración 10. Componente léxico

- **Círculo:** Representa el valor de un símbolo especial (como operadores, separadores, etc.). Se pueden interpretar también como análogos a símbolos terminales de la GLC.



### Ilustración 11. Símbolo Especial

- **Rectángulo:** Representa una estructura sintáctica que tiene su propio diagrama de sintaxis. Son similares a los símbolos no terminales de las GLC.



### Ilustración 12. Símbolo no-terminal

## Ejemplos de Diagramas de Sintaxis

A continuación, se mostrará un ejemplo sencillo extraído del trabajo de investigación de Sergio Gálvez Rojas, ingeniero informático de la Universidad de Málaga.

Es importante destacar que este siguiente ejemplo ha sido hecho en base al lenguaje de programación Modula-2, lenguaje de programación utilizado en la década de los 80s, por lo que ciertas estructuras vistas a continuación pueden variar de los lenguajes de programación de alto nivel de abstracción actuales.

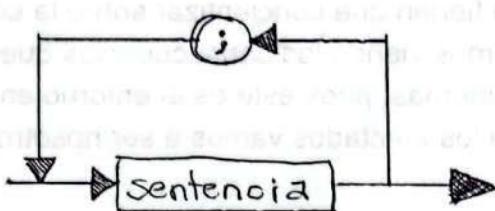


Ilustración 13. Diagrama de secuencia de sentencias.

En el ejemplo anterior, "se especifica que una secuenciaDeSentencias está formada por una sentencia, o bien, por una sentencia seguida del token (símbolo terminal) ';' y a secuencia de sentencias". (Gálvez, s.f. p.3). Esto representa básicamente la sintaxis que muchos lenguajes de programación de alto nivel siguen en la que, después de escribir una sentencia, se agrega un punto y coma ';' al final de ésta para indicar que la sentencia ha terminado.

Además, como se puede observar en el ejemplo anterior, la expresión 'sentencia' es un símbolo no-terminal, el cual también se desglosa en su propio diagrama sintáctico, el cual, siguiendo el ejemplo del libre de Gálvez, se representa de la siguiente forma:

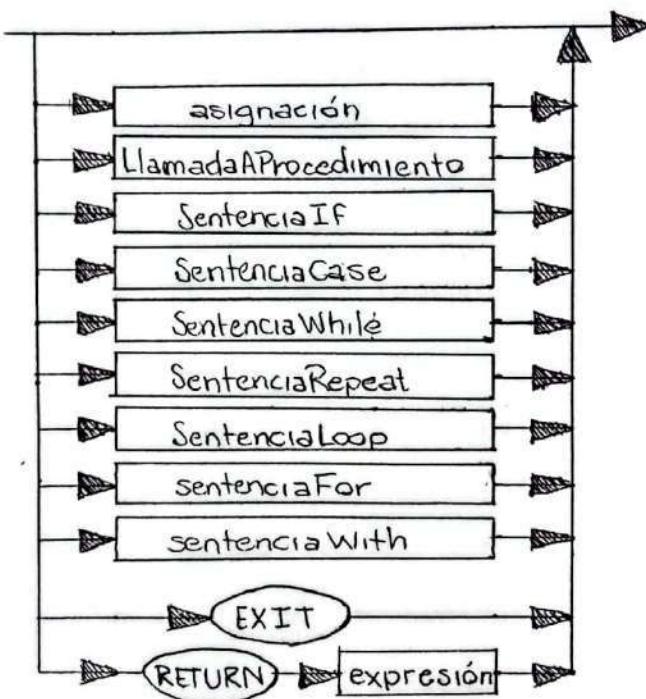


Ilustración 19. Diagrama de Sentencia.

A continuación, se podrá observar un ejemplo en donde el no terminal "secuenciaDeSecuencias" es utilizado en un diagrama de sintaxis para las sentencia case:

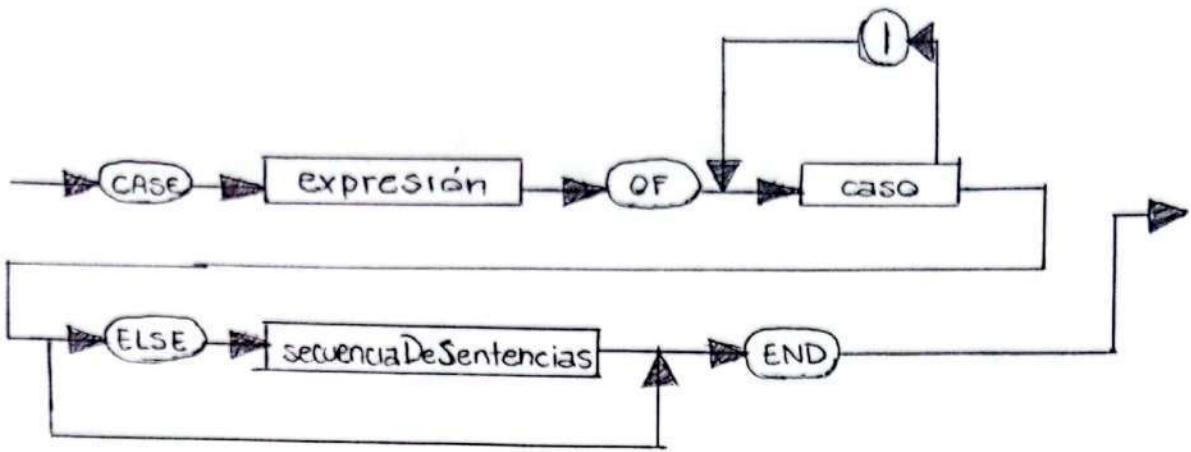


Ilustración 15. Diagrama de sentencia Case.

Como se pudo observar, los símbolos terminales pertenecientes a la gramática del lenguaje de Modula-2 como 'CASE', 'OF', 'ELSE' y 'END' son palabras reservadas que se encuentran dentro de elipses, mientras que los símbolos terminales especiales, como ';' o '1' se representan dentro de círculos. Por otro lado, los no terminales son claramente representados dentro de rectángulos.

También pueden usarse para representar la sintaxis de lenguajes naturales como el español o inglés como se muestra en el siguiente ejemplo:

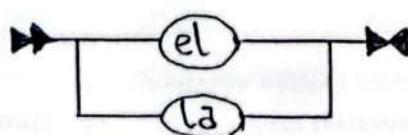


Ilustración 16. Diagrama del terminal artículo

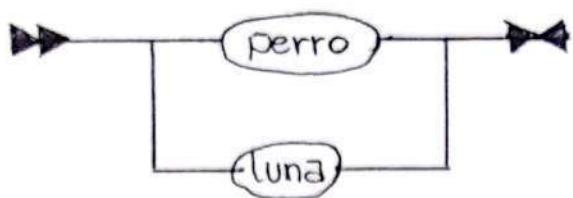


Ilustración 17. Diagrama del terminal sustantivo.



Ilustración 18. Diagrama del terminal verbo.

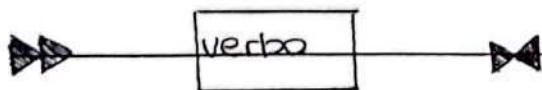


Ilustración 19. Diagrama del no-terminal predicado.

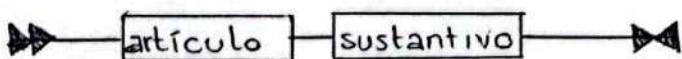


Ilustración 20. Diagrama del no-terminal sujeto.

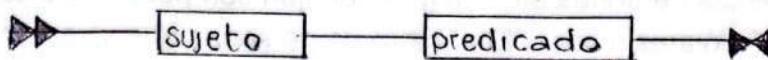


Ilustración 21. Diagrama del no-terminal **Frase**.

Como se pudo observar en el ejemplo anterior, el uso de diagramas de sintaxis no sólo se limita a los lenguajes formales, sino también en lenguajes naturales. Sin embargo, se reconoce su mayor utilidad y practicidad para la representación de lenguajes de programación/formales.

Así, gracias a los diagramas de sintaxis, se puede representar de manera gráfica la sintaxis que sigue determinado lenguaje y resulta una alternativa viable con respecto a la notación estandarizada BNF, que lejos de ser un recurso gráfico en sí, es más bien un recurso escrito que a muchas personas se les puede dificultar un poco su análisis.

Retomando este ejemplo, se puede escribir la frase "El perro corre" y gracias al diagrama, se puede analizar si esa frase pertenece o no a la gramática española,

*[Handwritten signatures]*

y si cumple con la sintaxis del español; sin embargo, si se escribe la frase "El corre luna", con el diagrama y la estructura sintáctica que posee el español, esa entrada será rechazada y se determinará que la frase no pertenece a la gramática del español; por el contrario, si se escribe la frase "El luna corre", se cumple con la sintaxis del español porque sigue la estructura → **sujeto** → **predicado** establecida, sin embargo, en su nivel semántico, esto será incorrecto porque el significado de la frase no tiene sentido.

El estudio del nivel semántico de los lenguajes corresponde a la etapa de análisis semántico, el cual no compete aún en esta actividad, sin embargo, es importante tenerlo presente porque forma parte íntegra de los tres análisis (léxico, sintáctico, semántico) de las gramáticas y, por tanto, de los lenguajes.

# Conclusiones

Esta monografía ahonda en el concepto de las Gramáticas Libres de Contexto (GLC) o gramáticas de tipo dos en la jerarquía de Chomsky. Destacar la importancia de estas gramáticas y estudiarlas es imprescindible para el estudio de los lenguajes formales, especialmente en su fase de análisis sintáctico pues, por ejemplo, los lenguajes de programación utilizados en la actualidad provienen de GLCs ya que éstas permiten recursividad.

Entonces, convergemos en la opinión de que gran parte materia amerita el estudio exhaustivo de los lenguajes de programación, es importante comprender el rol que juegan las GLCs en los lenguajes formales y, además, conocer los árboles de derivación y las diagramas de sintaxis como las herramientas que utilizan para hacer representaciones gráficas de la sintaxis de los lenguajes durante su etapa de análisis sintáctico.

~~GRACIAS~~  
~~Alejandra M. Cárdenas~~

Estamos de acuerdo en que las GLCs son imprescindibles en los lenguajes formales al ser parte de la mayoría de éstos, especialmente en los de programación de alto nivel con los que más estamos relacionados, y comprendemos los diferentes procesos de análisis por los que pasa un lenguaje con ayuda de las GLCs en un compilador o intérprete para desglosar, analizar, verificar y traducir las instrucciones de alto nivel en instrucciones de bajo nivel.

Nos damos cuenta de que es un proceso que se compone de diversas fases y que tiene una gran ingeniería por detrás, la cual nos compete totalmente a nosotros como futuras ingenieras en sistemas.



## Bibliografía

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2008). Teoría de Autómatas, Lenguajes y Computación 3/E. ADDISON WESLEY.

De Extremadura Departamento De Ingeniería De Sistemas Informáticos Y Telemáticos, U. (2008). Teoría de autómatas y lenguajes formales.

<https://dehesa.unex.es/handle/10662/2367>

Gramáticas Libres de Contexto. (2004, 29 marzo). Universidad Zaragoza.

Recuperado 20 de septiembre de 2023, de

[https://webdiis.unizar.es/asignaturas/LGA/material\\_2003\\_2004/4\\_LIC\\_GIC\\_ADPD.pdf](https://webdiis.unizar.es/asignaturas/LGA/material_2003_2004/4_LIC_GIC_ADPD.pdf)

Reyes, A. (2022, 31 agosto). Árboles de Derivación (*parse trees*). pfafner.github.io.

<https://pfafner.github.io/tc2022/aulas/Aula13.pdf>

Un Profe de Informática. (2020, 8 mayo). Lenguajes y autómatas - Módulo 2.2 (Árboles de derivación) [Vídeo]. YouTube.

<https://www.youtube.com/watch?v=68PQ6FBZq88>

Neso Academy. (2017, 29 marzo). Derivation tree (Left & right derivation trees) [Vídeo]. YouTube. <https://www.youtube.com/watch?v=u4-rpIIV9NI>

Easy Theory. (2021, 28 julio). What is a parse tree? + example [Vídeo]. YouTube. <https://www.youtube.com/watch?v=a1OLDU1QAfw>

Craig'n'Dave. (2018, 5 abril). AQA A'Level BNF and Syntax diagrams [Vídeo]. YouTube. <https://www.youtube.com/watch?v=x1gGInKNCRw>

IBM documentation. (s. f.).

<https://www.ibm.com/docs/es/ds8900/9.3.0?topic=commands-understanding-syntax-diagrams>

IBM documentation. (s. f.-b). <https://www.ibm.com/docs/es/db2/11.1?topic=sql-how-read-syntax-diagrams>

Durán Jacobo, M. A. & TecNM Campus Hopelchén. (s. f.). Unidad 6.- Introducción a los Lenguajes formales. *ITS HOPELCHEN*.

Graña Gil, J. (2000). Técnicas de análisis sintáctico robusto para la etiquetación del lenguaje natural. <https://ruc.udc.es/dspace/handle/2183/12358>

*BB* *doce*  
*A. M. G.*

Centro Universitario UAEM, Texcoco. Autómatas y Lenguajes Formales.  
Disponible en: <http://ri.uaemex.mx/bitstream/handle/20.500.11799/34043/secme-16209.pdf?sequence=1>

Gálvez, S. G. R., Tinaquero, D. T. F., Guevara, A. G. P., & Carrillo, A. L. C. L. (s. f.).

*GENERACIÓN COMPLETA DE COMPILADORES MEDIANTE DIAGRAMAS DE  
SINTAXIS EXTENDIDOS.*

<https://info.iaia.lcc.uma.es/lcc/publicaciones/LCC829.pdf>

Millán, J. A. J. (2009). *Compiladores y procesadores de lenguajes.*

<https://elibro.net/es/ereader/itcelaya/33847>

Martínez López, F. (2015). Teoría, diseño e implementación de compiladores de Lenguajes. Madrid, RA-MA Editorial. Recuperado de <https://elibro.net/es/ereader/itcelaya/106460>

Equipo editorial de IONOS. (2020b). Compilador e intérprete: definición y diferencias. IONOS Digital Guide. <https://www.ionos.mx/digitalguide/paginas-web/desarrollo-web/compilador-e-interprete/>

SSL UTN-FRT. (2020, 3 septiembre). *Clase 28 Diagramas sintaxis BNF sin caratula*

[Vídeo]. YouTube. <https://www.youtube.com/watch?v=UVz8IJHNX2A>



Tecnológico Nacional de México en Celaya

Fecha de entrega:  
27/09/23

PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

Carrera: Ingeniería en Sistemas Computacionales

Nombre de la Asignatura: Lenguajes y Autómatas II

Docente: Ricardo González González

Practica 2 'Implicaciones de los contextos dentro de una gramática':

VIDEO: <https://youtu.be/R70K1o5xdMY>

## Introducción

En el presente documento se abarcarán distintos puntos relacionados directamente con las gramáticas libres de contexto (que ahora en adelante nos referiremos a esta por sus siglas GLC) que corresponde a una gramática tipo dos de acuerdo con la jerarquía de Chomsky, esta misma generara los **lenguajes independientes del contexto** que a efectos prácticos la mayoría de los lenguajes de programación entran en esta categoría (existen algunas excepciones).

Las GLC están directamente ligados al proceso de análisis sintáctico de un compilador, mientras que las gramáticas sensibles al contexto estarán ligadas al proceso de análisis semántico.

Se hace la mención que debido a que son temas mas bien presentativos se optó por realizar un video para que quede de forma mas clara los conceptos que se verán en la presente práctica.

Se abarcará un ejemplo práctico en el que se demuestre de una manera entendible y fácilmente abordable por cualquier persona lo que es una GLC mediante el uso de una baraja Poker.

Posteriormente se ejemplificará mediante varios casos de la vida real lo que seria un contexto y también se plantearan algunos ejemplos aterrizados sobre la vida académica y el lenguaje de programación SQL,



Tecnológico Nacional de México en Celaya

Fecha de  
entrega:  
27/09/23

PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

ligado con este mismo también se argumentará la relación de dicho contexto con una gramática que definirá una lenguaje formal.

Por último se mostrarán algunos árboles de derivación y se hará mención su utilidad a la hora del análisis sintáctico.



Tecnológico Nacional de México en Celaya

Fecha de entrega:  
27/09/23

PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## Objetivo / Competencia

- Explicar que son las gramáticas libres de contexto mediante un ejemplo práctico y sencillo de comprender.
- Definir qué es una gramática y mencionar su utilidad a la hora de definir un lenguaje formal mediante una gramática.
- Mostrar la importancia y utilidad de los árboles de derivación al realizar el análisis sintáctico.

**Autores:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## Fundamento

Las gramáticas libres de contexto son un concepto fundamental en la definición de lenguajes formales, especialmente en los lenguajes de programación, los cuales deben tener una sintaxis predecible y precisa para facilitar su uso por diversos programadores sin necesidad de que éstos sean expertos en el lenguaje y su estructura.

Dichas gramáticas están ligadas directamente a los lenguajes formales que son imprescindibles para crear lenguajes de programación o notaciones científicas ya que tienen la propiedad que a diferencia de los lenguajes naturales presentan poca o nula ambigüedad, esto es necesario para una comprensión única tanto por parte de cualquier programado como también de parte del compilador.

Para tratar un tema como este, es necesario tener en cuenta el concepto de contexto, el cual está altamente ligado a este tipo de gramáticas pues, al no depender de éstos, tiene una funcionalidad muy diferente a las gramáticas dependientes del contexto que se mantienen sus reglas sujetas al contexto por el que se encuentran rodeadas. El contexto vendría siendo lo que le da una significancia adicional a una instrucción dada, dependiendo si una gramática es libre o no de contexto esta misma tendrá o no relevancia para su uso.

Las GLC están detrás de la gran mayoría de los lenguajes de programación de alto nivel, lenguajes que los ingenieros en sistemas, especialmente desarrolladores, utilizan en su día para trabajar y crear programas, por ello, en esta práctica se profundiza en su concepto, uso, relación con el contexto (o no relación) y se destaca la importancia de éstas para los sistemas computacionales.



Tecnológico Nacional de México en Celaya

Fecha de entrega:  
27/09/23

PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## Requisitos Básicos

Para la presente práctica únicamente se requiere de lo siguiente:

- Libreta, hojas blancas o cualquier medio de anotación
- Lápiz, Pluma o cualquier medio de escritura
- Se recomienda (pero no es totalmente necesario) un compás y una regla.
- Conocimientos sobre gramáticas libres de contexto
- Noción sobre contextos
- Conocimientos sobre elaboración de árboles de derivación, así como también los conceptos de árbol de derivación a la izquierda y árbol de derivación a la derecha.

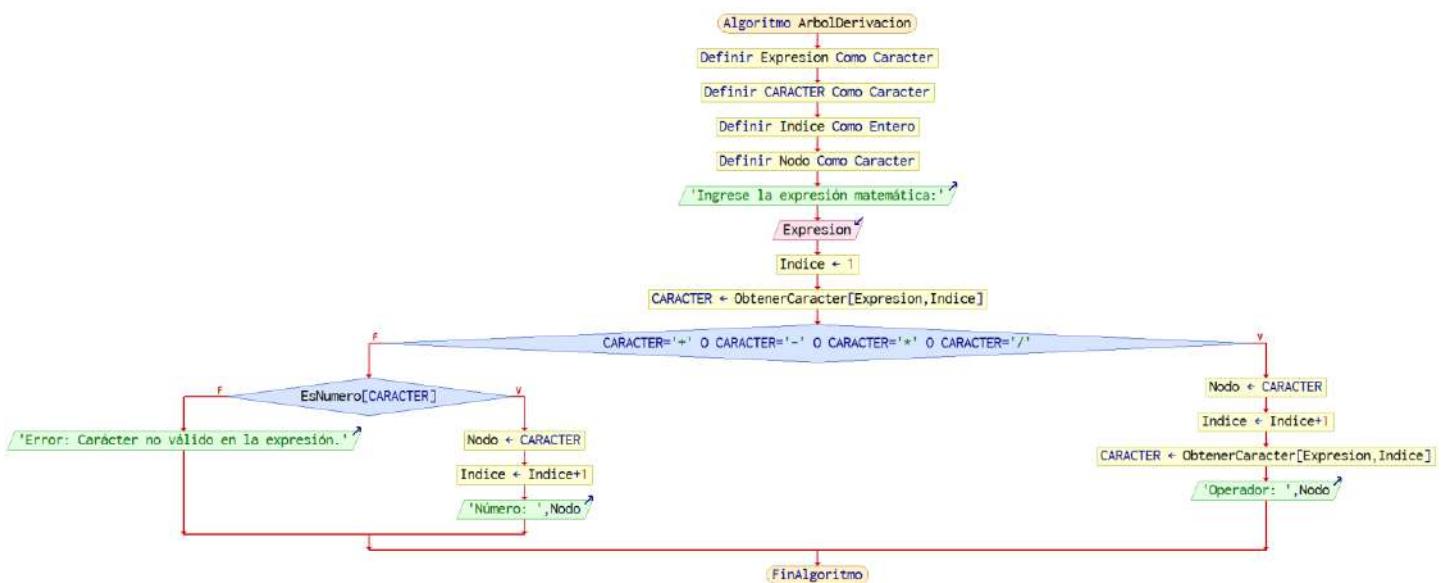
PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

**Autores:**  
Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## Desarrollo

## DIAGRAMA DE FLUJO

Para realizar un árbol de derivación





PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## PSEUDOCÓDIGO

Algoritmo ArbolDerivacion

Definir Expresion Como Cadena

Definir Caracter Como Caracter

Definir Indice Como Entero

Definir Nodo Como Caracter

Escribir "Ingrese la expresión matemática:"

Leer Expresion

Indice = 1

Caracter = ObtenerCaracter(Expresion, Indice)

Si Caracter = "+" O Caracter = "-" O Caracter = "\*" O Caracter = "/" Entonces

Nodo = Caracter

Indice = Indice + 1

Caracter = ObtenerCaracter(Expresion, Indice)

Escribir "Operador: ", Nodo

Sino

Si EsNumero(Caracter) Entonces

Nodo = Caracter

Indice = Indice + 1

Escribir "Número: ", Nodo

Sino



Tecnológico Nacional de México en Celaya

Fecha de  
entrega:  
27/09/23

PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

Escribir "Error: Carácter no válido en la expresión."

FinSi

FinSi

FinAlgoritmo



PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## EXPLICACION

### ¿QUÉ SON LAS GRAMÁTICAS LIBRES DE CONTEXTO ?

En palabras sencillas, una gramática libre de contexto es una gramática tipo dos que, a diferencia de las expresiones regulares, admite recursividad.

Estas mismas vendrían siendo un conjunto de reglas que describen la estructura sintáctica de un lenguaje. Se dice que son “libres de contexto” ya que dichas reglas se aplicaran independientemente del contexto en el que se encuentren las palabras en una oración.

De manera de ejemplo se puede plantear un juego de cartas como vendría siendo el póker. Supongamos que queremos describir las reglas para formar una mano ganadora.

Podemos utilizar una gramática libre de contexto para esto:

- Una mano de póker se compone de cinco cartas.
- Las cartas pueden ser de cuatro palos diferentes: corazones, diamantes, picas y tréboles.



- Cada carta tiene un valor numérico del 2 al 10 o puede ser una carta especial como “As”, “Rey”, “Reina” o “jota”

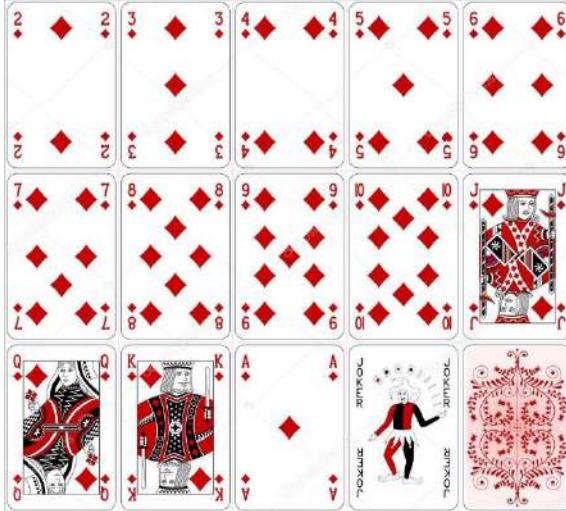


PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López





Entonces para formar una mano ganadora, se pueden tener diferentes combinaciones, como por ejemplo “par” (dos cartas del mismo valor), un trio (tres cartas del mismo valor), un “full house” (un par y un trio), “escalera” (cinco cartas con valores consecutivos), “Poker” (cuatro cartas del mismo valor / dos pares).

En el juego, las reglas para formar una mano de póker son independientes de las cartas específicas que se usen o del orden tanto que se presenten o sean barajeadas. Esto tiene similitud a como las gramáticas libres de contexto describen la estructura de un lenguaje sin depender del contexto específico de las palabras en una oración.



PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## ¿QUÉ ES UN CONTEXTO?

Es el entorno o situación que rodea una palabra, frase o una acción y que influye en su significado o interpretación.

Respecto a una GLC la estructura gramatical de una oración se determina únicamente en función de la gramática y no lo que rodea alguna oración, mientras que en una gramática dependiente del contexto si toma en cuenta dichas palabras para determinar su estructura.

**Ilustración:** Imagina que estas en una fiesta de disfraces. Y tienes una foto tuya donde llevas puesto un atuendo en particular. Sin el contexto que fue una fiesta, alguien que vea la foto podría pensar que vistes de una manera extraña en tu día a día cotidiano o que tienes gustos de vestimenta cuestionables. Pero cuando se conoce el contexto se comprende que dicho disfraz era parte de la diversión y/o temática del evento.



Otro ejemplo:



Tecnológico Nacional de México en Celaya

Fecha de entrega:  
27/09/23

PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López



Imagina que un día recibes un mensaje “Estoy en camino”, sin ningún contexto puede significar muchas cosas o a la par no significar nada. Podrías tomar la decisión de bloquear el emisor del mensaje pero si sabes de ante mano que estas esperando a un amigo en la parada del autobús y te envía dicho mensaje, entonces comprenderás que tu amigo ya está próximo a llegar a dicho lugar (o al menos en teoría).

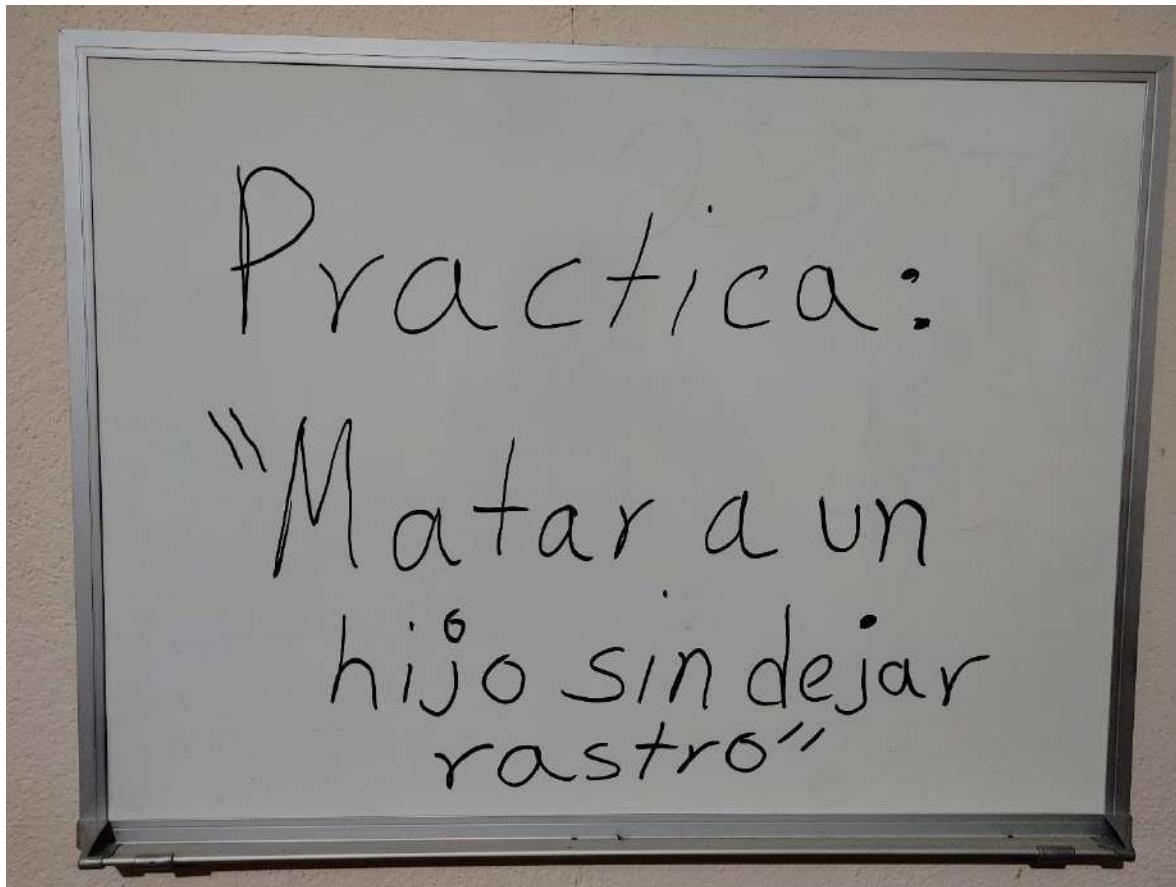


PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

Otro ejemplo un poco más extremo: Llegas a tu salón de clases un poco tarde, por lo tanto no recuerdas la clase ni que se está haciendo solo ves lo siguiente en el pizarrón.



Sin contexto dicho mensaje es cuanto menos raro, sin embargo recuerda que estás en la clase de sistemas operativos y que en realidad lo que necesitas hacer es matar a un proceso hijo en la terminal de Linux y que no te devuelva un mensaje de confirmación.

**PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II****Autores:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

Por último un ejemplo relacionado para ver la diferencia entre una gramática libre de contexto y una dependiente del contexto:

Por ejemplo como procesaríamos la palabra “baja” en español.

Con una GLC podemos definir una regla que diga “baja” es un verbo en tercera persona del singular, y se aplicara en cualquier contexto.

Sin embargo, en una gramática dependiente del contexto podríamos necesitar el contexto para determinar si al mencionar “baja” se refiere a “ella baja las escaleras”, “la baja temperatura”, “la baja probabilidad”, “la baja tarifa monetaria”. En este caso se vuelve esencial el contexto.

**PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II****Autores:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

**¿ SIRVE DICHO CONTEXTO A LOS PROPÓSITOS DE UNA GRAMÁTICA QUE DEFINA UN LENGUAJE FORMAL ?**

Totalmente, el contexto es fundamental ya que ayudara a establecer reglas precisas y comprender la estructura del lenguaje formal a que determina cómo se aplican las reglas gramaticales y cómo se interpretan las expresiones en ese lenguaje. El grado de importancia que tendrá sobre una gramática dependerá en su tipo y el lenguaje formal que se está definiendo.

Teniendo los siguientes casos:

**Gramáticas Libres de Contexto (GLC):** En muchos casos, las GLC son suficientes para definir lenguajes formales, como el lenguaje de programación de computadoras. Estas gramáticas son independientes del contexto, lo que significa que se aplican de la misma manera en cualquier lugar dentro del lenguaje, es decir no nos importara el contexto. Esto facilita la generación de análisis sintácticos automáticos y la comprensión de la estructura de programas informáticos.

**Gramáticas Dependientes del Contexto (GDC):** Algunos lenguajes formales pueden requerir una descripción más precisa del contexto, lo que hace que las GDC sean más apropiadas. Por ejemplo, en la definición de un protocolo de comunicación, puede ser necesario tener en cuenta el estado actual de la comunicación para comprender la estructura de los mensajes.



PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## ÁRBOLES DE DERIVACIÓN Y SU APORTACIÓN AL ANÁLISIS SINTACTICO

En palabras simples, los árboles de derivación son grafos conexos con un único nodo raíz, estos nos permitirán representar gráficamente el como se puede derivar cualquier cadena a partir de un símbolo distinguido es decir nos permiten visualizar como las reglas gramaticales generan estructuras jerárquicas en un lenguaje, siendo de gran importancia para ver como las palabras y frases se combinan de acuerdo con dichas reglas gramaticales de modo que se ajustan a las gramáticas libres de contexto.

Para nosotros los humanos dichos recursos son fáciles de interpretar gracias a que los podemos visualizar de una manera visual las derivaciones realizadas, pero el mayor beneficio se encuentra relacionado al compilador ya que gracias a que dicho árbol es la estructura de datos que representa el programa fuente, le facilitará la traducción del programa fuente a código ejecutable ya que dicho proceso será realizado por funciones naturalmente recursivas.

Veamos algunos ejemplos:

### EJEMPLO 1

Tenemos la siguiente GLC:

- a.  $S \rightarrow ASB$
- b.  $S \rightarrow d$
- c.  $S \rightarrow b$
- d.  $A \rightarrow aaBB$
- e.  $A \rightarrow aA$
- f.  $B \rightarrow dcD$

Y se nos pide derivar la siguiente cadena: abddcd

PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

## Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

Para realizar dicho árbol nuestras proceder deberán ser el siguiente:

1° S (aplicamos la 1ra)

2° ASB (a la mas izquierda aplicamos la 5 para obtener nuestra primera a)

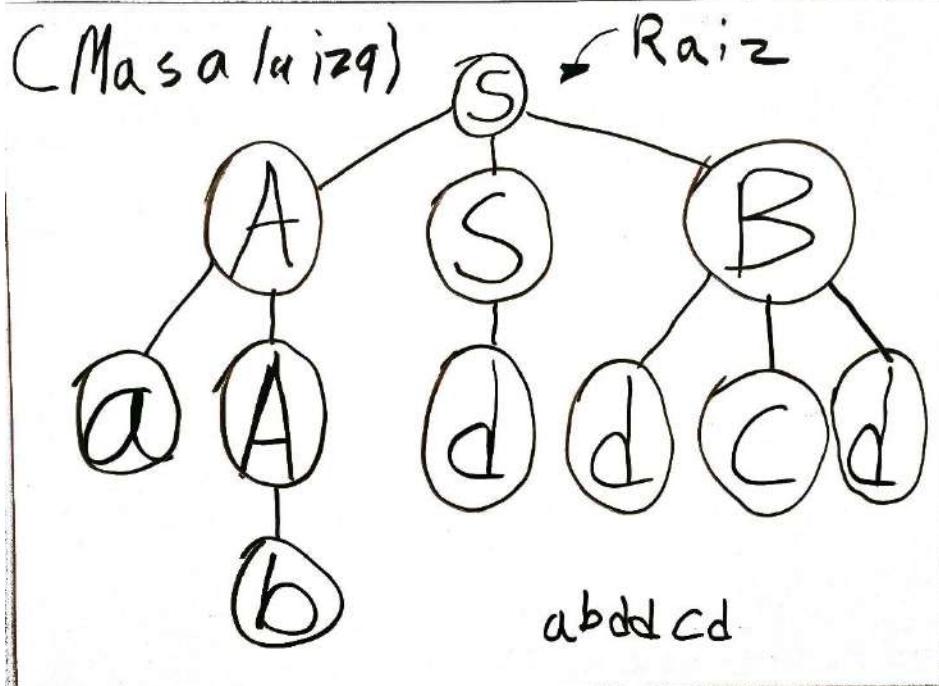
3° aASB (a la mas izquierda no terminal aplicamos la 3 para cambiar la A por la b)

4° abSB (aplicamos a S la producción numero 2 para obtener d)

5° abdB (aplicamos la producción 6 para obtener la parte que nos falta)

6° abddcd (hemos llegado a nuestra derivación pedida)

Entonces si lo queremos representar como árbol su resultado es:





PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## EJEMPLO 2

Contamos con la siguiente GLC:

- 1)  $S \rightarrow zMNz$
- 2)  $M \rightarrow aMa$
- 3)  $M \rightarrow z$
- 4)  $N \rightarrow bNb$
- 5)  $N \rightarrow z$

Y se nos pide derivar la siguiente cadena: zazabzbz

Para realizar dicho árbol nuestras derivaciones deberán ser los siguientes:

- 1) Comenzar con el nodo raíz S.
- 2) S (aplicamos la 1, principalmente porque no tenemos de otra).
- 3) zMNz (Al mas izquierda es decir M le aplicamos la 2da).
- 4) zaMaNz (ahora a M aplicamos la 3 para obtener la z faltante en ese bloque).
- 5) zazaNz (a N aplicamos la 4ta para obtener una aproximación similar al paso 3).
- 6) zazaMbz (a M le aplicamos la 5 para terminar).
- 7) zazabzbz

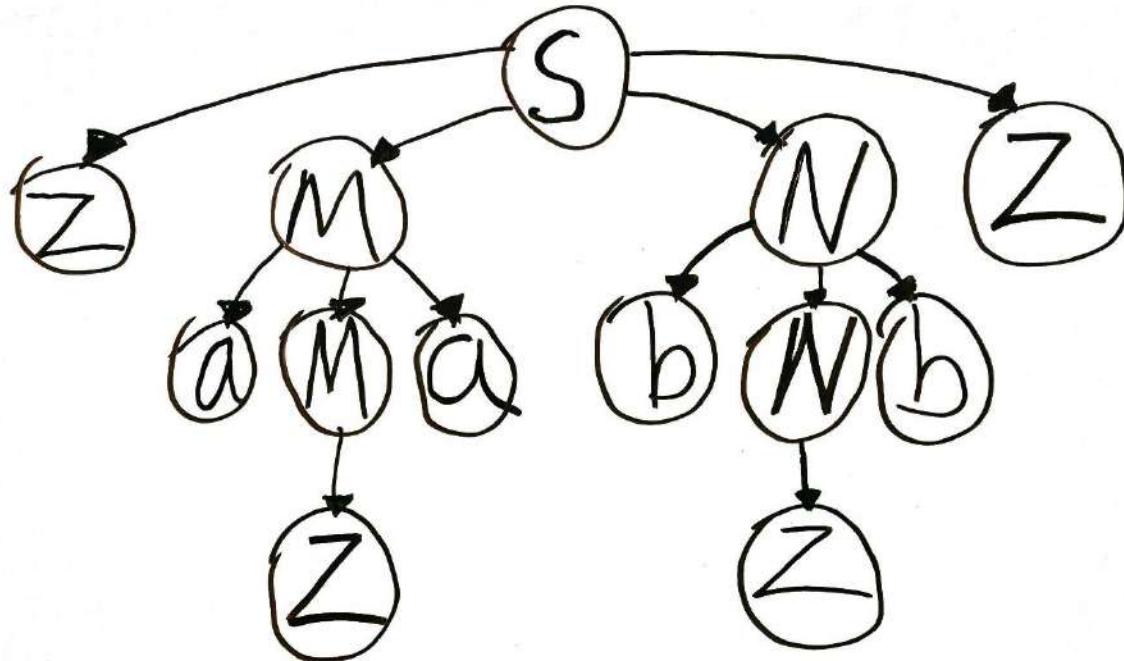
PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

## Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

*[Handwritten signatures]*

Entonces si lo queremos representar como árbol su resultado es:



Derivar: zazabzbz



PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

### EJEMPLO 3

Contamos con la siguiente GLC:

- 1)  $E \rightarrow E + E$
- 2)  $E \rightarrow E \times E$
- 3)  $E \rightarrow id$

Y se nos pide derivar la siguiente cadena:  $id + id \times id$

Para realizar dicho árbol nuestro proceder deberá ser el siguiente:

- 1) Comenzar con nuestro símbolo inicial  $E$ .
- 2)  $E$  (aplicar la 1 para establecer la estructura general).
- 3)  $E + E$  (aplicamos a la primera  $E$  la 3 para obtener el  $id$ ).
- 4)  $Id + E$  (sobre la  $E$  aplicamos la 2 para establecer el signo de multiplicación).
- 5)  $Id + E \times E$  (a ambas  $E$  aplicamos la 3 para obtener el  $id$ , solo respetando primero el de la izquierda).
- 6)  $Id + id \times id$

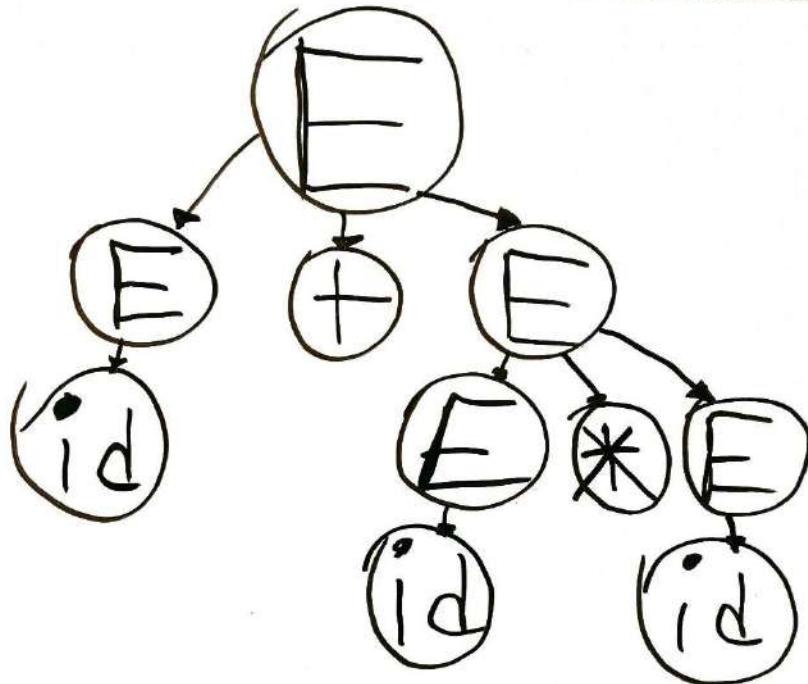
Entonces si lo queremos representar como árbol su resultado es:



PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López



Derivar:  $\text{id} + \text{id} * \text{id}$



PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## EJEMPLO 4

Contamos con la siguiente GLC:

- 1)  $S \rightarrow aAS|aSS| \epsilon$
- 2)  $A \rightarrow SbA|ba$

Y se nos pide derivar la siguiente cadena: aabaa

Para realizar dicho árbol nuestro proceder deberá ser el siguiente:

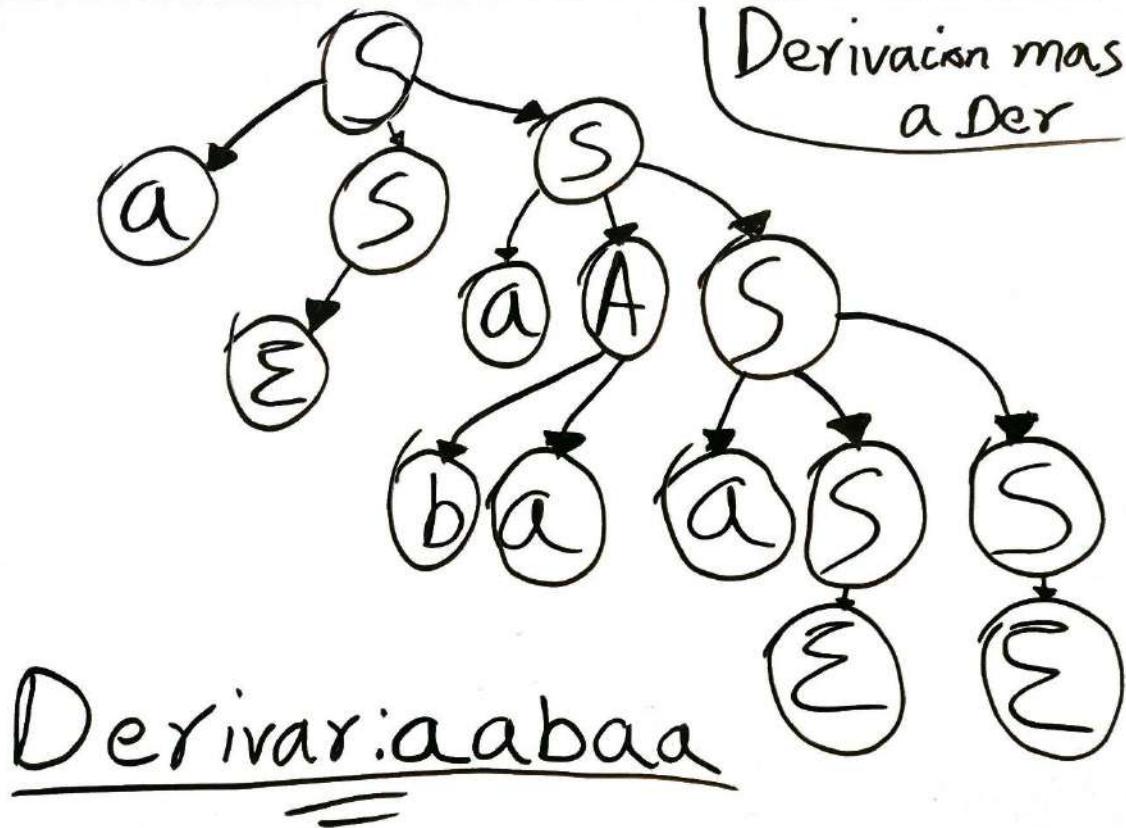
- 1) Comenzar con el símbolo inicial S
- 2) Usamos la única producción para S pero elegimos la 2da opción ya que es la que mas se ajusta a nuestro caso
- 3) Ass (ya que estamos haciendo un árbol de derivacion derecha entonces a la última s le aplicamos la primera producción opción 1).
- 4) aSaAS (ahora aplicamos de nuevo la producción 1 opción 2)
- 5) aSaAass (ya que no tenemos nada de utilidad en las últimas 2 s, entonces usamos la producción para hacerlos vacíos).
- 6) aSaAa (ahora en la A usamos la producción 2).
- 7) aSabaa (la S utilizamos también la cadena vacía para anularlo)
- 8) aabaa

Entonces si lo queremos representar como árbol su resultado es:



PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

Autores:  
Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López





Tecnológico Nacional de México en Celaya

Fecha de entrega:  
27/09/23

PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

**Autores:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## Bitácora de Incidencias

BITÁCORA	
INCIDENCIA	SOLUCIÓN
Al momento de realizar la grabación del video al compañero le dio pantalla azul y no quería dejar de grabar	Simplemente sacarlo de la llamada e intentar de nuevo.

**PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II****Autores:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## Conclusión

Gracias a esta práctica hemos comprendido la importancia de las Gramáticas Libres de Contexto (GLC) en la descripción de la estructura de lenguajes formales. Se comprende que las GLC son un conjunto de reglas gramaticales que permiten generar secuencias de símbolos en lenguajes, donde cada regla especifica cómo se pueden combinar estos símbolos.

Uno de los conceptos fundamentales que hemos abordado es el de "contexto". En nuestra vida cotidiana, el contexto es esencial para comprender y comunicarnos eficazmente. Hemos visto ejemplos de contexto en situaciones tan simples como una conversación, donde las palabras pueden adquirir significados diferentes según el contexto en el que se utilicen.

Hemos relacionado este concepto de contexto con las GLC al comprender que estas gramáticas se centran en la estructura de las secuencias de símbolos y no tienen en cuenta el contexto en el que se utilizan. Esto las hace adecuadas para describir la sintaxis de lenguajes formales, como los lenguajes de programación, donde las reglas gramaticales deben ser precisas y predecibles.

Comprender todo lo que se realizó en esta práctica es esencial para quienes deseen trabajar en el campo de la lingüística, la programación o la teoría de la computación.



Tecnológico Nacional de México en Celaya

Fecha de entrega:  
27/09/23

PRACTICA DE LABORATORIO  
LENGUAJES Y AUTOMATAS II

**Autores:**

Alma Gabriela Ponce Morales  
Anthony Gómez Cabañas  
Cristhian Alberto Ortega Hernández  
Miguel Ángel Ruiz López

## Referencias Bibliográficas

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2008). Teoría de Autómatas, Lenguajes y Computación 3/E. ADDISON WESLEY.

Neso Academy. (2017, 29 marzo). Derivation tree (Left & right derivation trees) [Vídeo]. YouTube. <https://www.youtube.com/watch?v=u4-rpIIV9NI>

Easy Theory. (2021, 28 julio). What is a parse tree? + example [Vídeo]. YouTube. <https://www.youtube.com/watch?v=a1OLDU1QAfw>

Martínez López, F. (2015). Teoría, diseño e implementación de compiladores de Lenguajes. Madrid, RA-MA Editorial. Recuperado de <https://elibro.net/es/ereader/itcelaya/106460>

Millán, J. A. J. (2009). Compiladores y procesadores de lenguajes. <https://elibro.net/es/ereader/itcelaya/33847>

## Notación BNF

Por sus siglas en inglés Backus-Naur Form (BNF) o traducido como notación de Backus-Naur (o a veces también presentado como Backus-Naur formalism o Backus Normal Form) es un metasintaxis o metalinguaje (decir un lenguaje que se usa para hablar sobre otro lenguaje) que es utilizado para expresar las gramáticas libres de contexto (GLC) siendo un método formal para describir la sintaxis de los lenguajes de programación (que son obtenidos a partir de las GLC).

Fue introducido por John Backus un diseñador de lenguajes de programación de IBM y Peter Naur en 1960. Aunque el planteamiento original de traducir la estructura de un lenguaje con reglas de escritura se remonta al trabajo del gramático indio Panini (aproximadamente 460 a. C.) para la descripción de la estructura de palabras de idioma sanscrito eh de ahí porque se ha sugerido renombrarla la forma Panini-Backus.

Su estructura es: name :: = expansion

Siendo sus características:

*Agradecimientos*

**Notación Concisa:** Es simple y compacta para describir las reglas gramaticales de un lenguaje de programación. Consiste en reglas de producción que indican cómo se forman las construcciones del lenguaje.

**Reglas de producción:** Constan de un símbolo no terminal (como "`<expresión>`") seguido de ":" = " y luego una expresión que describe como se forma ese no terminal. Dicha expresión puede incluir otros no terminales, terminales (símbolos concretos) y operadores como "/" para alternativas o "(" para agrupar.

A continuación se plantea una estructura <sup>BNF</sup> para describir la estructura de una dirección de correo electrónico:

`<dirección-correo> ::= <nombre-usuario> "@" <dominio>`  
`<nombre-usuario> ::= <letra> (<letra> | <dígito> | "-" | "_")*`  
`<dominio> ::= <nombre-dominio> ".> <extensión>`  
`<nombre-dominio> ::= <letra> (<letra> | <dígito> | "-")*`  
`<extensión> ::= <letra> (<letra> | <dígito>)*`  
`<letra> ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z"`  
`<dígito> ::= "0" | "1" | ... | "9"`

~~AP~~ ~~des~~  
~~A~~ ~~M~~ ~~C~~

En el ejemplo anterior, se definió la estructura de un correo electrónico. Las reglas de producción indican que una dirección de correo consta de un <nombre-usuario> seguido de "@" y un <dominio>.

El <nombre-usuario> se compone de letras, dígitos, guiones "-" o guiones bajo "\_" mientras que el <dominio> se compone de un <nombre-dominio> seguido de un punto y una <extención>. El <nombre-dominio> puede contener letras, dígitos y guiones y la <extención> está formada por letras y dígitos.

# ~~Atas~~ Criothian Alberto Ortega Hernández Bachus-Naur Form (BNF)

~~Atas~~  
Atas

La forma Bachus-Naur (BNF) es una gramática libre de contexto que es ampliamente utilizada por los desarrolladores de lenguajes de programación para definir las reglas de sintaxis de un lenguaje. John Backus, un desarrollador de lenguajes de programación, desarrolló esta notación originalmente para documentar IAL, una de las primeras implementaciones de Algol. Después, Peter Naur contribuyó a perfeccionar el enfoque de Bachus.

BNF utiliza una variedad de símbolos y expresiones que permiten escribir reglas de producción. Una regla de producción simple es:

**<digito>** ::= 0|1|2|3|4|5|6|7|8|9

Los corchetes angulares (**<>**) son usados para denotar un símbolo no terminal. Si un símbolo no terminal aparece en el lado de recho de la regla de producción, significa que habrá otra regla de producción que define su reemplazo.

## Características

**Expresividad:** La BNF permite expresar de manera precisa y concreta las reglas de producción en un lenguaje formal.

**Sintaxis clara y legible:** La BNF usa una sintaxis sencilla y fácil de entender. Usa símbolos como **::=** para indicar "se define como" y **|** para indicar "o". También permite el uso de paréntesis para agrupar elementos y especificar repetición.

**Descripción de reglas de producción:** Las reglas de producción se definen asignando símbolos no terminales a combinaciones de símbolos terminales y no terminales.

~~Alejandro M. Sánchez~~

Oporte para recursividad: Se permite la definición de reglas de producción recursivas, lo que significa que un símbolo no terminal puede aparecer en la parte derecha de su propia regla de producción.

### Ejemplo

Sintaxis para un domicilio

```
<Address> ::= <House number><Street name><Town name>
              <City name><Country><Postcode> | <House number>
              <Street name><City name><Country><Postcode>

<Postcode> ::= <Area code><Street code>
<Area code> ::= <City prefix><digit> | <City prefix><digit>
                  <digit>
<Street name> ::= <Name><Street Type>
<Flat number> ::= <character><digit>
<House number> ::= <number> | <digit><number>
<number> ::= <digit> | <digit><number>
<Name> ::= <string>
<Street Type> ::= <string>
<City prefix> ::= <string>
<Street code> ::= <string>
<Town name> ::= <string>
<City name> ::= <string>
<Country> ::= <string>
<string> ::= <character> | <character><string>
<character> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X
                  Y|Z|
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

# BNF

BNF, también conocida como "Forma normal de Backus" representa un método muy sintético ampliamente empleado para definir de manera precisa la estructura y las sintaxis de lenguajes de programación informática, conjuntos de comandos e instrucciones, la presentación de documentos y protocolos de comunicación. Se utiliza para esculpir la gramática de un lenguaje o sistema en términos comprensibles, lo que permite a programadores y diseñadores de sistemas comprender y comunicar las reglas fundamentales que rigen la construcción de software, el intercambio de datos y la comunicación entre sistemas informáticos.

## Características

Una especificación BNF se configura como un conjunto de reglas de derivación que típicamente se presenta en el siguiente formato

- Los no terminales se representan con la notación  $::=$  seguidos de una expresión.
- La expresión está compuesta por una o más secuencias de símbolos
- Para indicar múltiples opciones de secuencia, se utiliza la barra vertical  $\langle \rangle$ .

- Los símbolos que no se encuentran en el lado izquierdo de " $\cdot \cdot \Rightarrow$ " se consideran terminales.
- Los símbolos que aparecen en el lado izquierdo forman parte de la regla y no son considerados terminales.

## EJEMPLOS

$\langle \text{Expresión} \rangle ::= \langle \text{termino} \rangle \mid \langle \text{expresión} \rangle ^+ \text{ } \langle \text{termino} \rangle$

$\langle \text{terminos} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{termino} \rangle ^* \langle \text{factor} \rangle$

$\langle \text{Factor} \rangle ::= \langle \text{número} \rangle \mid (^{'} \langle \text{expresión} \rangle ^{'} )$

$\langle \text{número} \rangle ::= '0' \mid '1' \mid '2' \mid \dots \mid '9'$

Se puede pasar a explicar lo anterior mostrado:

$\langle \text{expresión} \rangle$  representa una expresión matemática.

$\langle \text{termino} \rangle$  puede representar un término en la expresión, que puede ser un número o una operación de multiplicación.

$\langle \text{factor} \rangle$  puede representar un factor en un término, que puede ser un número o una expresión entre paréntesis.

$\langle \text{número} \rangle$  define dígitos dentro de un rango válido del 0 al 9.

La gramática anterior permite construir expresiones válidas, como " $2 + 3 * (4 + 5)$ ", donde los números se combinan con operadores de suma y multiplicación.

Ruiz López Miguel Ángel

~~A. Ruiz M. López~~

# BNF

La notación Backus-Naur (BNF) es una forma formal de describir la estructura de un lenguaje de programación. Fue desarrollada por John Backus y Peter Naur en 1960. Funciona como un conjunto de reglas que se utilizan para explicar cómo se deben escribir las instrucciones en un lenguaje formal o, más específicamente, lenguajes de programación de manera clara y precisa.

Sirve como un estándar durante el análisis sintáctico que muestra el comportamiento de una estructura sintáctica del lenguaje. Además, garantiza que todos los programadores sigan las mismas reglas al escribir código, lo que facilita la lectura y comprensión del código por parte de otros programadores.

En otras palabras, representa sintaxis de lenguajes de programación y sólo se utiliza con gramáticas libres de contexto (GLC).

## Metacaracteres/metásimbolos

- <> Paréntesis angulares: Encierran ~~memotécnicas~~ que representan las No-Terminales de las GLC.
- ::= Paamayim Nequdotayim igual: Representa la derivación de un elemento a otro.

- | barra vertical: Separa las distintas alternativas de escribir un no-terminal
- { } LLaves: Encierran secuencias que se repiten.  
Ej.  $A \rightarrow A\beta | \lambda$        $\langle A \rangle ::= \{\beta\}$
- [ ] Corchetes: Encierran secuencias optativas  
Ej.  $A \rightarrow \beta | \lambda$        $\langle A \rangle ::= [\beta]$
- ( ) Encierran varias secuencias alternativas con prefijos y/o sufijos comunes  
Ej.  $w(x | y | z) u$

Ejemplo:

$\langle \text{Identificador} \rangle ::= \langle \text{Letra} \rangle \langle \text{Resto} \rangle | \langle \text{Guion} \rangle \langle \text{Resto} \rangle$

$\langle \text{Letra} \rangle ::= a | b | \dots | z$

$\langle \text{Digito} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

$\langle \text{Resto} \rangle ::= \{ \langle \text{Letra} \rangle | \langle \text{Digito} \rangle | \langle \text{Guion} \rangle \}$

$\langle \text{Guion} \rangle ::= -$



## Referencias Bibliográficas

UNIVERSIDAD EUROPEA DE MADRID. (s. f.). ANÁLISIS LÉXICO. Cartagena99. Recuperado 21 de septiembre de 2023, de [https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2\\_M4\\_U2\\_T1.pdf](https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2_M4_U2_T1.pdf)

Bavera, F., Nordio, D., & Arroyo, M. (s. f.). JTLEX un Generador de Analizadores Léxicos Traductores. sedici. Recuperado 21 de septiembre de 2023, de [http://sedici.unlp.edu.ar/bitstream/handle/10915/23090/Documento\\_completo.pdf?sequence=1&isAllowed=y#:~:text=Un%20analizador%20l%C3%A9xico%20es%20un,tokens%20correspondientes%20y%20sus%20atributos.](http://sedici.unlp.edu.ar/bitstream/handle/10915/23090/Documento_completo.pdf?sequence=1&isAllowed=y#:~:text=Un%20analizador%20l%C3%A9xico%20es%20un,tokens%20correspondientes%20y%20sus%20atributos.)

[Agu01] J. Aguirre, V. Grinspan, M. Arroyo, J. Felippa, G. Gomez, "JACC un entorno de generación de procesadores de lenguajes" Anales del CACIQ 01, 2001.

Ortega, A., & De la Cruz, M. (2008). Construcción de un analizador léxico para ALFA con Flex. arantxa. Recuperado 21 de septiembre de 2023, de [http://arantxa.ii.uam.es/~mdlcruz/docencia/compiladores/2007\\_2008/lexico\\_07\\_08.pdf](http://arantxa.ii.uam.es/~mdlcruz/docencia/compiladores/2007_2008/lexico_07_08.pdf)

Hernandez, G. S. Q. (s. f.). GENERADOR DE ANALIZADORES LÉXICOS (LEX & FLEX). prezi.com. <https://prezi.com/p/skykfvgrxm7/generador-de-analizadores-lexicos-lex-flex/>

De València Escola Tècnica Superior d'Enginyeria Informàtica, U. P. (2020, 9 noviembre). Flex. Desarrollo de un analizador léxico usando flex. <https://riunet.upv.es/handle/10251/10071>

Parr, T. J., & Quong, R. W. (1995). ANTLR: A predicated-LL(K) Parser generator. Software - Practice and Experience, 25(7), 789-810. <https://doi.org/10.1002/spe.4380250705>

Bavera, F. (2002, 1 octubre). JTLEX un generador de analizadores léxicos traductores. <http://sedici.unlp.edu.ar/handle/10915/23090>

Scott, M. L. (2006). *Programming Language Pragmatics* (2<sup>a</sup> ed.). Elsevier. <https://doi.org/10.1016/b978-0-12-374514-9.x0001-8> (Obra original publicada en 2000)

Mogensen, T. (2007). *Basics of compiler design* (2<sup>a</sup> ed.). s.n.]. <http://hjemmesider.diku.dk/~torbenm/Basics/> (Obra original publicada en 2000)

Levine, J., Brown, D., & Mason, T. (1992). *Ilex & yacc* (2<sup>a</sup> ed.). O'Reilly Media. <http://www.nylxs.com/docs/lexandyacc.pdf>

Fischer, C. N., Cytron, R. K., & LeBlanc Jr, R. J. (2010). *Crafting a compiler*. Pearson Education. <https://studylib.net/doc/26053844/crafting-a-compiler-by-charles-n.-fischer--ron-k.-cytron-...>

*[Handwritten signatures]*

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers, principles, techniques, and tools* (2<sup>a</sup> ed.). Pearson Education.  
<https://drive.google.com/file/d/0B1MogsyNAsj9eIVzQWR5NWVTSVE/view?resourcekey=0-zoBDMPzTafr6toxDuQLNUg> (Obra original publicada en 1986)

Vilar Torres, J. M. (2010, October 7). *Analizador léxico*. Universitat Jaume I.  
[https://repositori.uji.es/xmlui/bitstream/handle/10234/22657/II26\\_analisis\\_lexico.pdf?sequence=1](https://repositori.uji.es/xmlui/bitstream/handle/10234/22657/II26_analisis_lexico.pdf?sequence=1)

Raffo Lecca, E. (2007, mayo). *Programación genérica en C++*. Redalyc.  
Recuperado 22 de septiembre de 2023, de  
<https://www.redalyc.org/pdf/816/81610112.pdf>

Rosenfeld, R. (2022, 26 julio). *Introducción a la verificación de programas*.  
<http://portalreviscien.uai.edu.ar:9999/ojs/index.php/RAIA/article/view/198>

Isaac Computer Science. (s. f.). Isaac Computer Science.

[https://isaaccomputerscience.org/concepts/dsa\\_toc\\_bnf?examBoard=all&stage=all](https://isaaccomputerscience.org/concepts/dsa_toc_bnf?examBoard=all&stage=all)

Theory of Computation: Backus-Naur form - Wikibooks, Open Books for an Open world. (s. f.). [https://en.wikibooks.org/wiki/A-level\\_Computing/AQA/Paper\\_1/Theory\\_of\\_computation/Backus-naur\\_form](https://en.wikibooks.org/wiki/A-level_Computing/AQA/Paper_1/Theory_of_computation/Backus-naur_form)

Greyrat, R. (2022b, julio 5). Notación BNF en el diseño del compilador – Barcelona Geeks. <https://barcelonageeks.com/notacion-bnf-en-el-diseno-del-compilador/>

Dutoit, Y. P. D. (s. f.). Notación\_de\_Backus-Naur: Definición de Notación\_de\_Backus-Naur y sinónimos de Notación\_de\_Backus-Naur (español). sensagent - 2005-2015.  
[https://diccionario.sensagent.com/Notaci%C3%B3n\\_de\\_Backus-Naur/es-es/](https://diccionario.sensagent.com/Notaci%C3%B3n_de_Backus-Naur/es-es/)

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2008). Teoría de Autómatas, Lenguajes y Computación 3/E. ADDISON WESLEY.

De Extremadura Departamento De Ingeniería De Sistemas Informáticos Y Telemáticos, U. (2008). Teoría de autómatas y lenguajes formales.  
<https://dehesa.unex.es/handle/10662/2367>

Gramáticas Libres de Contexto. (2004, 29 marzo). Universidad Zaragoza.  
Recuperado 20 de septiembre de 2023, de  
[https://webdiis.unizar.es/asignaturas/LGA/material\\_2003\\_2004/4\\_LIC\\_GIC\\_ADPN\\_D.pdf](https://webdiis.unizar.es/asignaturas/LGA/material_2003_2004/4_LIC_GIC_ADPN_D.pdf)



Reyes, A. (2022, 31 agosto). Árboles de Derivación (parse trees). pfafner.github.io.

<https://pfafner.github.io/tc2022/aulas/Aula13.pdf>

Un Profe de Informática. (2020, 8 mayo). Lenguajes y autómatas - Módulo 2.2 (Árboles de derivación) [Vídeo]. YouTube.

<https://www.youtube.com/watch?v=68PQ6FBZq88>

Neso Academy. (2017, 29 marzo). Derivation tree (Left & right derivation trees) [Vídeo]. YouTube. <https://www.youtube.com/watch?v=u4-rpIIV9NI>

Easy Theory. (2021, 28 julio). What is a parse tree? + example [Vídeo]. YouTube. <https://www.youtube.com/watch?v=a1OLDU1QAfw>

Craig'n'Dave. (2018, 5 abril). AQA A'Level BNF and Syntax diagrams [Vídeo]. YouTube. <https://www.youtube.com/watch?v=x1gGInKNCRw>

IBM documentation. (s. f.).

<https://www.ibm.com/docs/es/ds8900/9.3.0?topic=commands-understanding-syntax-diagrams>

IBM documentation. (s. f.-b). <https://www.ibm.com/docs/es/db2/11.1?topic=sql-how-read-syntax-diagrams>

Durán Jacobo, M. A. & TecNM Campus Hopelchén. (s. f.). Unidad 6.- Introducción a los Lenguajes formales. *ITS HOPELCHEN*.

Graña Gil, J. (2000). Técnicas de análisis sintáctico robusto para la etiquetación del lenguaje natural. <https://ruc.udc.es/dspace/handle/2183/12358>

Centro Universitario UAEM, Texcoco. *Autómatas y Lenguajes Formales*. Disponible en: <http://ri.uaemex.mx/bitstream/handle/20.500.11799/34043/secme-16209.pdf?sequence=1>

Gálvez, S. G. R., Tinaquero, D. T. F., Guevara, A. G. P., & Carrillo, A. L. C. L. (s. f.).

*GENERACIÓN COMPLETA DE COMPILADORES MEDIANTE DIAGRAMAS DE SINTAXIS EXTENDIDOS.*

<https://info.iaia.lcc.uma.es/lcc/publicaciones/LCC829.pdf>

Millán, J. A. J. (2009). *Compiladores y procesadores de lenguajes*.

<https://elibro.net/es/ereader/itcelaya/33847>



Martínez López, F. (2015). Teoría, diseño e implementación de compiladores de Lenguajes. Madrid, RA-MA Editorial. Recuperado de <https://elibro.net/es/ereader/itcelaya/106460>

Equipo editorial de IONOS. (2020b). Compilador e intérprete: definición y diferencias. IONOS Digital Guide. <https://www.ionos.mx/digitalguide/paginas-web/desarrollo-web/compilador-e-interprete/>

SSL UTN-FRT. (2020, 3 septiembre). *Clase 28 Diagramas sintaxis BNF sin caratula*

[Vídeo]. YouTube. <https://www.youtube.com/watch?v=UVz8IJHNX2A>