

# INSTITUTO TECNOLÓGICO NACIONAL DE MÉXICO EN CELAYA.

Materia: Lenguajes y Autómatas II

Maestro: ISC. Ricardo González González

Alumnos:

Isaac Salvador Bravo Estrada 2003048

Guillermo Poasland Aguilar 20030737

Maria del Carmen Chávez Patiño 20030296

Luis Fernando Mendoza Javalera 1930536

## ► ACTIVIDAD 5 ◄

Fecha de entrega: 14 de Septiembre  
de 2024

EQUIPO NO. 3



(Sello)

(Firma)

(Firma)

(Firma)

## DEPARTAMENTO DE SISTEMAS COMPUTACIONALES E INFORMÁTICA

ASUNTO: **SOLICITUD DE ACTIVIDADES**

Celaya, Guanajuato, 17 /septiembre / 2024

### LENGUAJES Y AUTÓMATAS II

DOCENTE DESIGNADO: ISC. RICARDO GONZÁLEZ GONZÁLEZ

**SEMESTRE AGOSTO-DICIEMBRE 2024**

**ACTIVIDAD 5 (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.

### 3. ANÁLISIS SINTÁCTICO.

A. 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 :

- PRECEDENCIA DE OPERADORES
- ANALIZADOR SINTÁCTICO DESCENDENTE ( LL )
- ANALIZADOR SINTÁCTICO ASCENDENTE ( LR, LALR )
- DISEÑO Y ADMINISTRACIÓN DE UNA TABLA DE SÍMBOLOS.
- MANEJO DE ERRORES SINTÁCTICOS Y SU RECUPERACIÓN.
- GENERADORES DE CÓDIGO PARA ANALIZADORES SINTÁCTICOS: YACC, BISON ( REPASO Y REFUERZO DE LO REVISADO ANTES ).

#### 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

**¿ CÓMO FUNCIONA UN ANALIZADOR DESCENDENTE ?**

**¿ CÓMO FUNCIONA UN ANALIZADOR ASCENDENTE ?.**

**¿ CUÁLES SON LOS OBJETIVOS Y LAS FUNCIONES DE UN ANALIZADOR SINTÁCTICO ?**

A MODO DE PRÁCTICAS REALICE ESTE PUNTO Y ELABORE EJERCICIOS NECESARIOS CON LOS CUÁLES USTED DEMUESTRE

**¿ CÓMO FUNCIONA EL GENERADOR DE CÓDIGO PARA ANALIZADORES SINTÁCTICOS YACC ?**

**¿ CÓMO FUNCIONA EL GENERADOR DE CÓDIGO PARA ANALIZADORES SINTÁCTICOS BISON ?.**

**¿ CUÁLES SON LAS CARACTERÍSTICAS Y LAS FUNCIONES DE ESTOS DOS ANALIZADORES SINTÁCTICO ?**

**ELABORE UN PAR DE VIDEOS DONDE EXPONGA CÓMO EMPLEÓ ESTAS DOS HERRAMIENTAS. COLOQUE SU MATERIAL EN YOUTUBE O EN ALGUNA OTRA PLATAFORMA E INCLUYA LA LIGA DENTRO DE SU EXAMEN.**

**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.





(Firmas)

- B. ELABORE **UN VIDEO DE AL MENOS 35 MINUTOS**, DONDE A MODO DE VIDEO CONFERENCIA COMENTE Y EXPLIQUE LO DESARROLLADO TODOS LOS INCISOS ANTERIORES.

SI TIENE REGISTRADO UN EQUIPO, HAGA LA VIDEO CONFERENCIA CON LOS INTEGRANTES Y EXPLIQUEN LO REALIZADO EN ESTA ACTIVIDAD.

**NOTA:** LA LIGA DEL VIDEO DEBERÁ SER INTEGRADA AL PDF DE LA ACTIVIDAD, PARA QUE AL HACER CLIC EN ÉSTE SE REPRODUZCA. POR ÚLTIMO, NO OLVIDE OTORGAR LOS PRIVILEGIOS NECESARIOS PARA COMPARTIR CORRECTAMENTE ESTE MATERIAL.





(Sello)

(Sello)

(Sello)

(Sello)

## **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.





(Firmas)

**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	:	LAI <sub>II</sub> , LI MÁS EL GRUPO ( -A , -B, -C )
DOCUMENTO	:	A1-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 PERIODO SEMESTRAL EN CURSO: <b>AGO-DIC</b>

**EJEMPLO :**

SI EL TRABAJO SE SOLICITÓ EN EQUIPO.

2024-09-17\_TNM\_CELAYA\_LAI<sub>II</sub>-A\_A5\_EQUIPO\_99\_9999999\_PEREZ\_PEREZ\_JUAN\_AGO-DIC24.PDF

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

SI EL TRABAJO SE SOLICITÓ INDIVIDUALMENTE.

2024-09-17\_TNM\_CELAYA\_LAI<sub>II</sub>-A\_A5\_9999999\_PEREZ\_PEREZ\_JUAN\_AGO-DIC24.PDF





(Sello)

(Firma)

(Firma)

(Firma)

**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.





# INSTITUTO TECNOLÓGICO NACIONAL DE MÉXICO EN CELAYA.

Materia: Lenguajes y Autómatas II

Maestro: I.S.C. Ricardo González González

Alumnos:

Isaac Salvador Bravo Estrada 2003048

Guillermo Peasland Aguilar 20030737

María del Carmen Chávez Patiño 20030296

Luis Fernando Mendoza Javalera 1930536

“Monografía sobre Análisis Sintáctico”

► Punto 3.A ◀

Fecha de Entrega: 24 de Septiembre de  
2024.

EQUIPO NO.3

# Índice de temas

1. Precedencia de Operadores
  - Concepto
  - Tabla de precedencia de operadores
  - Precedencia de operadores en C
  - Precedencia de operadores en expresiones regulares y autómatas
  - Buenas prácticas
2. Analizador Sintáctico Descendente (LL)
  - Funcionamiento básico del analizador LL(1)
  - Ejemplo de reconocimiento de una sentencia
  - Gramáticas LL(1)
  - Ejemplo 1 de condiciones LL(1)
  - Conflictos primero/primero
  - Producción donde se derive de ambas alternativas una cadena vacía
  - Conflictos primero/siguiente
  - Construcción de la tabla de análisis LL(1)
3. Analizador Sintáctico Ascendente (LR, LALR)
  - Algoritmo de modificación del autómata
  - Construcción de las tablas de análisis LALR
  - Construcción de la tabla de acción
  - Descripción a detalle de la construcción de la tabla de análisis LALR
4. Diseño y administración de una tabla de símbolos
  - Funciones principales de la tabla de símbolos
  - Diseño de la tabla de símbolos
  - Administración de la tabla de símbolos
5. Manejo de errores sintácticos y su recuperación
  - Tipos de errores sintácticos
  - Estrategias de manejo de errores sintácticos
  - Herramientas para la recuperación de errores
6. Generadores de código para analizadores sintácticos: YACC y BISON
  - YACC (Yet Another Compiler Compiler)
    - i. Gramáticas BNF
    - ii. Árboles de análisis
  - BISON
    - i. Ventajas sobre YACC
  - Comparación entre YACC y BISON

# Precedencia de Operadores

La precedencia de operadores es un concepto fundamental en diversas ciencias, como la matemática, la programación y en la teoría de lenguajes formales. Establece el orden en que se evalúan las expresiones que contienen múltiples operadores, ya sean expresiones matemáticas, lógicas o aritméticas. Este concepto es crítico para que el código funcione correctamente y las máquinas (ya sean compiladores o intérpretes) puedan evaluar las expresiones de manera predecible y sin errores. En esta monografía analizaremos en profundidad la precedencia de operadores, sus reglas, su papel en la teoría de autómatas y gramáticas formales, y las buenas prácticas para el uso en la programación.

## Concepto

En la programación, la precedencia de operadores dicta el orden en que se ejecutan las operaciones dentro de una expresión que contiene múltiples operadores. Por ejemplo:

$$2 + 3 * 4$$

Los operadores tienen diferentes prioridades. En este caso, el operador de multiplicación (\*) tiene mayor precedencia que el operador de suma (+) por lo que la multiplicación se realiza primero, seguido por la suma. El resultado de esta expresión es 14,

ya que se resuelve de la siguiente manera

$$2 + 3 * 4$$

$$2 + 12$$

$$\underline{14}$$

Este concepto es clave para evitar las ambigüedades al evaluar expresiones. Si no existieran reglas claras sobre el orden en que deben ejecutarse los operadores, los resultados podrían variar y no ser predecibles.

### Tabla de precedencia de operadores

Los lenguajes de programación siguen reglas estrictas de precedencia de operadores. A continuación, se presenta una tabla de precedencia común que aplica a la mayoría de los lenguajes, aunque con algunas variaciones dependiendo del lenguaje.

Operador	Descripción
Paréntesis ( )	Tiene la mayor precedencia y permiten anular las reglas de precedencia natural.
Exponenciación **	En algunos lenguajes permiten esta operación con este operador.

Operador	Descripción
Multiplicación * División / Módulo %	Tienen la misma precedencia.
Suma + Resta -	Estos operadores se evalúan después de la multiplicación, división y módulo. Al igual que los operadores anteriores se evalúan de izquierda a derecha a menos que se utilicen parentesis.
Operadores de comparación <, >, <=, >=	Los operadores de comparación (como menor que <, mayor que >) tienen una precedencia menor que los operadores aritméticos.
Operadores de equidad == !=	Los operadores de equidad (igual == y distinto !=) comparan la igualdad o desigualdad entre 2 valores.
Operadores Logicos &&	Los operadores logicos AND && y OR    se evalúan al final, siendo AND de mayor precedencia que OR

Cuando dos operadores tienen la misma precedencia, se evalúan de izquierdo a derechas, salvo en casos específicos como la exponentiación, que puede ser evaluada de derecho a izquierda en ciertos lenguajes.

## Precedencia de operadores en C

El lenguaje de programación C sigue reglas similares para la precedencia de operadores. En C, entender correctamente este concepto es crucial para evitar errores en la evaluación de expresiones. Un error común es olvidar que la multiplicación, división y modulo tienen mayor precedencia que la suma y la resta. En una expresión como:

resultado = a \* b + c / q - b;

La multiplicación y la división se ejecutan primero y luego se realiza la suma y la resta. Si queremos alterar el orden, ponemos un parentesis para controlar la evaluación de las operaciones, como en:

resultado = (a \* b) + (c / q) - b;

El uso de parentesis no solo garantiza que las operaciones se ejecuten en el orden correcto, si no que también hace el código mas legibles y fácil de leer.

Otro ejemplo sería

$$\text{resultado} = (2+3)^* 4;$$

Aquí, los paréntesis fuerzan que se ejecute primero la suma  $2+3$ , antes de realizar la multiplicación. Esto genera un resultado de  $20$ , mientras que sin los parentesis, el resultado sería  $14$  debido a la precedencia natural de los operadores.

### Precedencia de Operadores en Expresiones Regulares y Automatas.

En la teoría de los lenguajes formales, la precedencia de operaciones también es importante, especialmente en la evaluación de expresiones regulares. Estas expresiones, que describen conjuntos de cadenas en lenguajes formales, utilizan operadores como la concatenación, la unión ( $\cup$ ) y la cerradura de kleene (\*), por ejemplo

$$(q_1 b)^* c$$

La cerradura de kleene (\*) tiene mayor precedencia que la concatenación. Esto significa que la repetición infinita se aplica primero a la unión de  $q_1 b$  y luego se concatena con  $c$ . Si no se respecta esta precedencia, el automata que procesa esta expresión no funcionaría correctamente. El uso adecuado de la precedencia permite construir automatos finitos deterministas o no deterministas que reconoscan lenguajes

específicos.

Otro ejemplo en lenguaje SQL la expresión:

A OR B AND C OR D

sería equivalente a esta expresión:

(A OR B) AND (C OR D)

## Buenas prácticas

El manejo correcto de la precedencia de operadores puede facilitar mucho la legibilidad y el mantenimiento de código. Algunas de las mejores prácticas incluyen:

### 1.- Uso de paréntesis siempre que sea necesario:

Aunque no siempre es obligatorio usarlos, los paréntesis clasifican el orden de las operaciones y eliminan la posibilidad de malentendidos. Aún si la precedencia es clara, los parentesis hacen que las expresiones sean más comprensible para otros.

### 2.- Dividir expresiones complejas: En lugar de expresiones complicadas en una sola línea, es recomendable en pasos más simples y almacenar los resultados parciales.

```
int factor1 = a * b;  
int factor2 = c / q;  
resultado = factor1 + factor2 - b;
```

3.- Evitar la dependencia excesiva de la precedencia: Aunque las reglas de precedencia están definidas, es buena práctica no depender de ellos de manera implícita. Un uso adecuado de paréntesis y expresiones bien estructuradas asegura que el código sea más claro y que no haya sorpresas en su ejecución.

Dependiendo del lenguaje que se utilice puede haber más o menos operadores para ser aplicados en distintas tareas, un ejemplo es Spread, que es un operador del lenguaje JavaScript, este permite distribuir los elementos de un iterable (como una cadena o un arreglo) en ubicaciones individuales dentro de una nueva estructura. Así mismo hay otros operadores en este lenguaje que tienen su propia precedencia.

## Conclusión

La precedencia de operadores es un concepto fundamental tanto en programación como en la teoría de automatas y lenguajes formales. Desde las reglas básicas que rigen los operadores aritméticos y lógicos hasta la implementación moderna de operadores o en la creación de un nuevo lenguaje o incluso la creación de compiladores o intérpretes. Conocer la precedencia asegura que las expresiones sean evaluadas de manera correcta y predecible.

Finalmente, el dominio de las reglas de preceden-

cia y el uso de parentesis para alterar este orden no solo mejora la eficiencia del código, sino tambien su claridad. Conocer y aplicar correctamente estas reglas previene de errores comunes y hace que el código sea más manejable, tanto en el contexto de lenguajes de programación como en la construcción de autómatas que dependen de gramáticas y expresiones regulares.



# MONOGRAFÍA TÉCNICA SOBRE EL ANALIZADOR SINTÁCTICO DESCENDENTE (LL)

## INTRODUCCIÓN

El análisis sintáctico descendente es una técnica fundamental dentro del campo del procesamiento del lenguaje natural y la compilación, utilizada para interpretar y validar la estructura de los programas informáticos. En este contexto, el análisis sintáctico LL (Left-to-right) es uno de los enfoques más utilizados para construir parsers que operan de manera eficiente en tiempo real. Este tipo de análisis se caracteriza para recorrer la cadena de entrada de izquierda a derecha, generando derivaciones más a la izquierda, lo que permite identificar si una secuencia de símbolos pertenece a un lenguaje formal específico.

El análisis LL se basa en una jerarquía de lenguajes libres de contexto (CFG por sus siglas en inglés) que puede ser representado mediante gramáticas. La subcategoría más común es LL(1), donde el número entre paréntesis indica cuántos símbolos de entrada se pueden mirar hacia adelante para decidir qué regla aplicar durante el análisis. Debido a su simplicidad y eficacia, las gramáticas LL(1) son ampliamente empleadas en compiladores y otros sistemas que requieren procesamiento sintáctico, aunque tienen ciertas limitaciones respecto a su poder expresivo comparadas con otros métodos como LR.

## FUNCIONAMIENTO BÁSICO DEL ANALIZADOR LL(1)

El analizador sintáctico descendente LL(1) utiliza una pila para hacer el análisis sintáctico, basándose en la arquitectura de **automata a pila**. Además utiliza una **tabla de análisis** que indica si la entrada es correcta y qué acción realiza con ella. El esquema básico se representa de la siguiente manera:

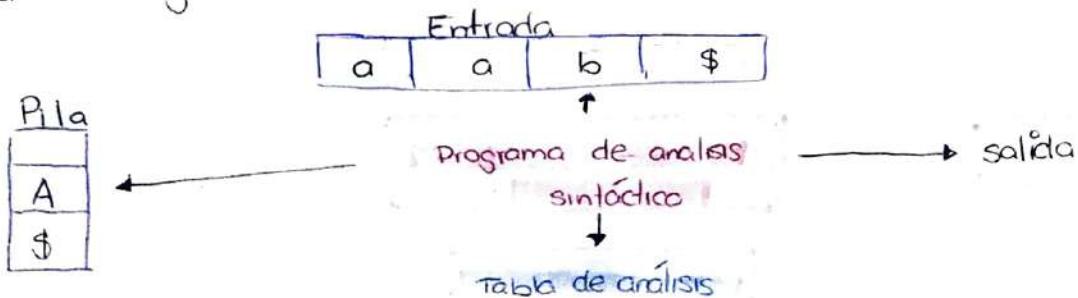


Figura 1. Funcionamiento.



El objetivo del analizador es reconocer si la frase pertenece a la gramática, para ello, utiliza una pila para almacenar los distintos símbolos gramaticales (terminales y no terminales) donde incorpora \$ en el inicio de la pila. En la entrada nos encontramos también \$ al final de la cadena de entrada, de tal forma que sino ha habido errores y, tanto en la entrada como en la pila solo queda \$, quiere decir que la frase ha sido reconocida como perteneciente a la gramática y termina el análisis con éxito. El programa de análisis busca en la tabla de análisis qué producción aplicar a partir de los símbolos que hay en la pila y en la entrada.

## EJEMPLO DE RECONOCIMIENTO DE UNA SENTENCIA.

Para poder entender cómo funciona el análisis sintáctico LL(1) utilizando el autómata o pila y la tabla de análisis, comenzaremos con una sentencia a reconocer y su tabla de análisis (posteriormente podremos ver cómo se construye la tabla y qué condiciones debe de cumplir).

### GRAMÁTICA:

$$\begin{aligned} S &\rightarrow (L) \mid id \\ L &\rightarrow SL' \\ L' &\rightarrow , SL' \mid \alpha \end{aligned}$$

Sentencia a reconocer: (a, b)

Tabla 1. Gramática.

Pasamos a la siguiente tabla de análisis, que es una matriz ( $M$ ) de dos dimensiones:

NO Terminales	Símbolos de entrada				
	(	)	id	,	\$
S	$S \rightarrow (L)$		$S \rightarrow id$		
L	$L \rightarrow SL'$		$L \rightarrow SL'$		
L'		$L' \rightarrow ,$		$L' \rightarrow SL'$	

Tabla 2. Análisis de símbolos.



## GRAMÁTICAS LL(1)

Tenemos que recordar que las gramáticas LL(1) forman parte de los analizadores sintáticos descendentes predictivos que son distintos a los recursivos. Los significados de las siglas son:

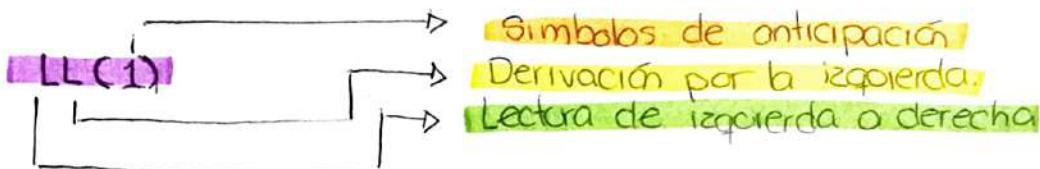


Figura 2. Significados de las siglas.

Y por lo tanto, en este tipo de gramáticas se recorre el árbol sintáctico de izquierda a derecha, y al hacerse el recorrido se selecciona las derivaciones más a la izquierda y son capaces de reconocer si la sentencia pertenece a la gramática con solo ver un símbolo anticipado.

Las gramáticas que son independientes del contexto (GIC), para poder analizarse con un analizador sintáctico descendente predictivo, LL(1) necesitan poder cumplir una serie de condiciones, ya que no todas las GIC son gramáticas LL(1). Es decir, Las gramáticas LL(1) son un subconjunto de las GIC y tienen las siguientes condiciones:

- 1: No puede ser ambigua: Es decir, no puede tener producciones con varias alternativas que comiencen por los mismos símbolos y si las hubiese habría que factorizar.
- 2: No puede ser recursivo por la izquierda.
- 3: Cuando haya producciones del tipo  $A \rightarrow \alpha | \beta$  se tienen que cumplir las siguientes condiciones:

Condiciones:

- 1: No puede haber conflictos entre los conjuntos PRIMERO de estas alternativas, es decir que no se podría o más bien no puede haber conflictos PRIMERO / PRIMERO.
- 2: Solo una opción puede derivar una cadena vacía ( $\lambda$ ) ya que todas las alternativas, como mucho pueden llevar a la cadena vacía ( $\lambda$ ), nunca dos al mismo tiempo. Esto ayuda a evitar la ambigüedad en el análisis.
- 3: No puede haber conflictos PRIMERO / SIGUIENTE ya que si una opción ( $\beta$ ) puede derivar la cadena vacía, entonces la otra opción ( $\alpha$ ) no debe empezar con ningún símbolo que esté en el conjunto SIGUIENTE de la regla. Esto asegura que no haya



confusión entre las posibles opciones cuando se espera un símbolo en particular.

Si la gramática elegida cumple estas condiciones de no ambigüedad y además las que se indican en el punto 3, se podría afirmar que la gramática es LL(1). Y como se puede observar, es un subconjunto relativamente pequeño de las GLC no ambiguas.

### EJEMPLO 1 DE CONDICIONES LL(1)

A partir de la gramática siguiente, vamos a verificar las condiciones de la misma para saber si es LL(1).

$$S \rightarrow (L) \mid id \quad L \rightarrow L, S \mid S$$

1: ¿Es ambigua la gramática?

No, puesto que la gramática no hay varias alternativas que comiencen igual.

2: ¿Es la gramática recursiva por la izquierda?

Sí, debido a la producción  $L \rightarrow L, S$ . Se aplica la siguiente regla:

$$A \rightarrow A\alpha \mid \beta, \text{ se resuelve } \Rightarrow A \rightarrow \beta A'$$

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

quedando la gramática como la siguiente:

$$S \rightarrow (L) \mid id \quad L \rightarrow SL' \quad L' \rightarrow, SL' \mid \lambda$$

3. ¿Hay producciones del tipo  $A \rightarrow \alpha \mid \beta$ ?

Si las hay, es necesario calcular los conjuntos PRIMERO y SIGUIENTE, para comprobar las siguientes 3 condiciones:

	PRIMERO	SIGUIENTE
S	C, id	\$, ..., )
L	C, id	,
L'	, , λ	)

Tabla 3. Producciones.

### CONFLICTOS PRIMERO / PRIMERO

Los conflictos PRIMERO / PRIMERO ocurren en gramáticas cuando dos producciones para un mismo no terminal comparten símbolos en sus conjuntos PRIMERO, lo que puede causar ambigüedades durante el análisis sintáctico. El objetivo es que cada producción comience con símbolos terminales diferentes, para que el analizador pueda decidir



Cuál producción aplicar al observar el primer símbolo de la entrada.

### EJEMPLOS:

1:  $S \Rightarrow (L) lid$ : No hay conflicto PRIMERO / PRIMERO porque el conjunto PRIMERO de la primera producción es  $\{\cdot\}$  y el de la segunda producción es  $\{lid\}$ . Como son diferentes, el analizador puede distinguir entre las dos producciones sin ambigüedad.

2:  $L' \Rightarrow SL' | \lambda$ : No hay conflicto PRIMERO / PRIMERO aquí porque el conjunto PRIMERO de la producción  $SL'$  es  $\{\cdot\}$  y el de la producción  $\lambda$  (cadena vacía) depende de los siguientes símbolos en la cadena (el siguiente PRIMERO después de la derivación vacía). Sin embargo, como el  $\cdot$  y  $\lambda$  son diferentes, tampoco hay conflicto en ese caso.

### PRODUCCIÓN DONDE SE DERIVE DE AMBAS ALTERNATIVAS LA CADENA VACÍA

Esto se refiere a que, en una reproducción, si dos alternativas permiten derivar la cadena vacía ( $\lambda$ ), puede haber ambigüedad. Si las dos alternativas derivan  $\lambda$ , el analizador no sabría cuál de ellas elegir, lo que podría causar problemas.

### EJEMPLO:

$L' \Rightarrow SL' | \lambda$ : Solo una de las alternativas (la segunda) permite derivar la cadena vacía ( $\lambda$ ). La primera producción  $SL'$  no deriva  $\lambda$  porque el símbolo  $,$  es un terminal, lo que impide que esa alternativa derive la cadena vacía.

### CONFLICTOS PRIMERO / SIGUIENTE

Los conflictos PRIMERO / SIGUIENTE ocurren cuando el conjunto PRIMERO de una producción que deriva  $\lambda$  (es decir, que puede llegar a la cadena vacía) tiene elementos en común con el conjunto SIGUIENTE del no terminal en cuestión. Esto puede causar ambigüedad porque el analizador no sabe si debe continuar expandiendo la producción o terminar la derivación en ese punto.

### EJEMPLO:

$L' \Rightarrow SL' | \lambda$ : Aquí  $B = \lambda$  (la alternativa que deriva la cadena vacía) y  $\alpha = SL'$  (la alternativa no vacía).

### ANALIZANDO:

• PRIMERO ( $\alpha$ ): Es PRIMERO de  $SL'$ , que es  $\{\cdot\}$ , ya que la terminal  $,$  es lo primero que aparece en esta alternativa.

• SIGUIENTE ( $L'$ ): Esto es el conjunto SIGUIENTE del no terminal  $L'$ . En el contexto de la gramática, este conjunto es  $\{\cdot\}$ , ya que probablemente después de  $L'$  puede aparecer un paréntesis derecho).



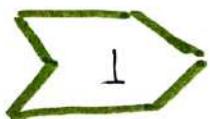
Como PRIMERO(SL) = {f, g} y SIGUIENTE(CL) = {f, g}, no hay elementos en común entre ambos conjuntos, por lo que **no hay conflicto PRIMERO / SIGUIENTE** en esta producción.

## CONSTRUCCIÓN DE LA TABLA DE ANÁLISIS LL(1)

La tabla de análisis sintáctico LL(1) es una matriz de dos dimensiones indexada por no terminales (N, columna de la izquierda en la tabla que hemos utilizado) y terminales (T, fila superior de la tabla, que incluye el símbolo \$). Por tanto accedemos a la tabla utilizando  $M[N, T]$ . En la tabla las entradas vacías son errores, que posteriormente se puede agregar las llamadas a funciones o procedimientos que avisan de algún error.

¿Cómo se construye esta tabla?

A partir de cada producción simbolizada por  $A \rightarrow \alpha$ , siendo  $\alpha$  cualquier cadena de terminales y no terminales, nos recorremos todas las producciones de la gramática y se hace lo siguiente:



Para cada terminal de PRIMERO ( $\alpha$ ), se añade  $A \rightarrow \alpha$  en  $M[A, t]$ , siendo  $t$  el terminal perteneciente al conjunto PRIMERO.



Si la cadena vacía está en PRIMERO ( $\alpha$ ), se añade  $A \rightarrow \lambda$ , para cada terminal ( $t$ ) de SIGUIENTE( $\alpha$ ) en  $M[A, t]$ . Esto se aplica también para \$, cuando aparece en SIGUIENTE( $\alpha$ ).

Figura 3. Construcción de la tabla

Para la creación de la tabla se hará con un Nuevo ejemplo:

## EJEMPLO 2 DE CONDICIONES LL(1)

Este ejemplo será con una gramática que no cumple las condiciones LL(1) para ver los efectos que esto produce en la construcción de la tabla. La gramática es la siguiente:

$S \rightarrow iCtS \mid iCs \mid S \rightarrow a \mid C \rightarrow b$



Se comienza a comprobar las condiciones LLC(1)

1: ¿Es ambigua la gramática?

Sí, puesto que no hay varias alternativas que comiencen igual. Lo corregimos, factorizando la gramática.

$S \rightarrow iC + SS'1a$     $S \rightarrow eS1\lambda$     $C \rightarrow b$

2: ¿Es la gramática recursiva por la izquierda?

No, puesto que no hay producciones que comiencen en el lado izquierdo por el mismo símbolo del lado derecho.

3: ¿Hay producciones del tipo  $A \rightarrow \alpha | \beta$ ?

Si las hay, es necesario calcular los conjuntos PRIMERO y SIGUIENTE, para comprobar las siguientes tres:

	PRIMERO	SIGUIENTE
S	i, a	\$, e
S'	e, $\lambda$	\$, e
C	b	$\tau$

Tabla 4. Conjuntos

### CONFLICTOS PRIMERO / PRIMERO

¿Hay conflictos PRIMERO/ PRIMERO en estas producciones?

Analicamos las dos producciones con varias alternativas:

•  $S \rightarrow iC + SS'1a$ : No tiene conflictos PRIMERO / PRIMERO, puesto que empiezan ambas por terminales ("i" y "a") y estos son diferentes.

•  $S \rightarrow eS1\lambda$ : No tiene conflictos PRIMERO / PRIMERO, puesto que "e" es distinto de la cadena vacía,  $\lambda$ .

### PRODUCCIÓN DONDE SE DERIVE DE AMBAS ALTERNATIVAS LA CADENA VACÍA

¿Hay alguna producción donde se derive de ambas alternativas de la cadena vacía?

No, puesto que no aparece  $\lambda$  en los dos conjuntos PRIMERO de las alternativas de la producción  $S \rightarrow eS1\lambda$ .

### CONFLICTOS PRIMERO / SIGUIENTE

¿Hay conflictos PRIMERO / SIGUIENTE en las producciones en las que de  $B$  se derive  $\lambda$ ?



En la producción  $S \rightarrow e \mid \lambda : \beta = \lambda$  y  $\alpha = es$

Analizamos PRIMERO ( $\alpha$ ), es decir  $\text{PRIMERO}(es) = \{e\}$ , mientras que  $\text{SIGUIENTE}(\lambda) = \text{SIGUIENTE}(S) = \{\$, e\}$ , por tanto  $e$  está en ambos  $\Rightarrow$  no cumple la condición ya que hay conflicto PRIMERO / SIGUIENTE.

### EJEMPLO DE CONSTRUCCIÓN DE TABLA LL(1)

A partir de la gramática anterior ahora veremos las consecuencias que produce el que una gramática no cumpla las condiciones y por tanto no sea una gramática LL(1). Partimos de la gramática a la que se han eliminado las ambigüedades.

$$S \rightarrow iC + SS' \mid \lambda \quad S \rightarrow e \mid \lambda \quad C \rightarrow b$$

De acuerdo con el algoritmo de creación de la tabla iremos analizando una a una las producciones de la gramática para construir la tabla, hasta que aparezca el problema que produce el conflicto detectado en la producción  $S \rightarrow e \mid \lambda$ .

EMPEZAMOS POR  $S \rightarrow iC + SS'$

No terminales	SÍMBOLOS DE ENTRADA					
	i	+	a	e	b	\$
S	$iC + SS'$					
S'						
C						

Tabla 5. Símbolos (1)

El conjunto PRIMERO ( $\alpha$ ) equivale a  $\text{PRIMERO}(iC + SS') = \{i\}$ , luego la entrada equivalente para  $M[A, +]$  será  $M[S, i] = S \rightarrow iC + SS'$ .

PARA  $S \rightarrow a$

No terminales	SÍMBOLOS DE ENTRADA					
	i	+	a	e	b	\$
S	$iC + SS'$		$S \rightarrow a$			
S'						
C						

Tabla 6. Símbolos (2)



# EDUCACIÓN

SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO  
NACIONAL DE MÉXICO

El conjunto PRIMERO ( $\alpha$ ) equivale a  $\text{PRIMERO}(\alpha) = \{a\}$ , luego la entrada equivalente para  $M[A, +]$ , será  $M[S, a] = S \rightarrow a$ .

**PARA  $S' \rightarrow \epsilon S$**

No terminales	SÍMBOLOS DE ENTRADA					
	i	+	a	e	b	\$
S	$S \rightarrow iCtSS'$		$S \rightarrow a$			
$S'$				$S' \rightarrow eS$		
C						

Tabla 7. Símbolos (3)

El conjunto PRIMERO ( $\alpha$ ) equivale a  $\text{PRIMERO}(\epsilon S) = \{\epsilon\}$ , luego la entrada equivalente para  $M[A, +]$ , será  $M[S', \epsilon] = S' \rightarrow \epsilon S$ .

**PARA  $S' \rightarrow \lambda$**

No terminales	SÍMBOLOS DE ENTRADA					
	i	+	a	e	b	\$
S	$S \rightarrow iCtSS'$		$S \rightarrow a$			
$S'$				$S' \rightarrow eS$	$S' \rightarrow \lambda$	
C					$C \rightarrow b$	

Tabla 8. Símbolos (4)

El conjunto PRIMERO ( $\alpha$ ) equivalente a  $\text{PRIMERO}(\lambda) = \{\lambda\}$ , estamos en la regla 2 y hay que ir a SIGUIENTE ( $S'$ ) =  $\{\$\}, \{\epsilon\}$ , por tanto  $M[A, +]$ , será  $M[S', \epsilon | \$] = S' \rightarrow \lambda$ .



## CONCLUSIÓN

El análisis sintáctico descendente LL ha demostrado ser una herramienta clave en el campo de la compilación y el procesamiento de lenguajes formales. A pesar de sus limitaciones frente a otros métodos más complejos, como los análisis LR, su simplicidad, legibilidad y eficacia en el procesamiento de gramáticas LL(1) lo hacen una opción ideal para construir parsers en tiempo real. Al ser capaz de analizar cadenas de entrada de izquierda a derecha y derivar secuencias desde el símbolo inicial, el análisis LL proporciona una solución eficiente para lenguajes que pueden ser descritos por reglas claras y predecibles.

Si bien las gramáticas LL(1) tienen restricciones que impiden su aplicación en ciertos lenguajes más complicados, su facilidad de implementación y la claridad que brindan en la fase de análisis sintáctico siguen siendo atractivas para desarrolladores de compiladores y sistemas operativos. El análisis sintáctico LL continúa siendo una técnica valiosa en el ámbito del análisis formal de lenguajes, brindando una base sólida para la creación de herramientas eficientes y funcionales en la programación y compilación.



# MONOGRAFÍA TÉCNICA SOBRE EL ANALIZADOR SINTÁCTICO ASCENDENTE (LR, LALR)

## INTRODUCCIÓN:

Al realizar análisis sintáctico es una de las etapas fundamentales en la construcción de compiladores, donde se verifica la estructura gramatical de las cadenas de entrada según las reglas de una gramática formal. Dentro de los analizadores sintácticos, existen dos enfoques principales: los analizadores descendentes y los ascendentes. En esta monografía nos enfocaremos en los analizadores sintácticos ascendentes, específicamente en las técnicas LR y LALR, que son ampliamente utilizadas debido a su eficiencia y capacidad para manejar una gran variedad de gramáticas.

Los analizadores LR (Left -to- Right derivation) son capaces de reconocer una amplia clase de lenguajes libres de contexto y son valorados por su capacidad para detectar errores de forma temprana.

El LALR (Look-Ahead LR) es una variación optimizada del LR que reduce el tamaño de las tablas de análisis sin sacrificar demasiada capacidad de análisis.

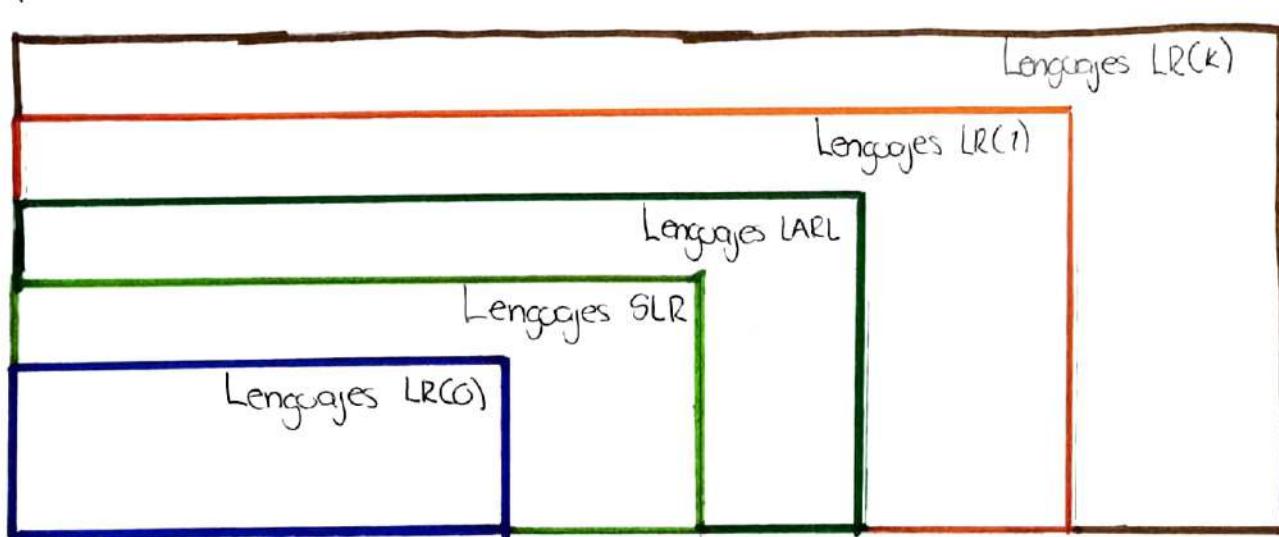


Figura 1. Diagramas.



Como se puede ver en el diagrama pasado no todas las gramáticas LR(1) pueden reconocerse con un analizador LALR, son por tanto un subconjunto de las LR(1).

En cuanto a los conflictos que pueden generarse con este tipo de gramáticas, estos son de reducción/reducción, nunca de desplazamiento/reducción.

## ALGORITMO DE MODIFICACIÓN DEL AUTÓMATA

Para entender como funciona el algoritmo de análisis sintáctico LALR, partimos del AFD obtenido con el LR(1), el cual debemos modificar siguiendo el siguiente proceso:

PRIMER PASO	Se seleccionan en el autómata LR(1) los estados con el mismo núcleo y se crea uno nuevo que es la unión de estos. El núcleo del nuevo estado será el núcleo común y como símbolo de anticipación tendremos la unión de todos los símbolos de anticipación de cada uno de los estados del conjunto.
SEGUNDO PASO	<p>Para cada estado del autómata modificado, hacemos lo siguiente:</p> <ul style="list-style-type: none"><li>• Si es un estado no modificado (para cada arista que salga de él). Si el estado modificado / llegada (unión de otros o ha desaparecido) entonces la arista irá al estado unión correspondiente al modificado.</li><li>• Si es un estado modificado. Todas las aristas que parten de este estado tienen que partir de la unión correspondiente y si el estado al que llegaban estas aristas es modificado entonces llegarán a la unión correspondiente.</li></ul>

Tabla 1. Modificación .

Como vemos, se fusionan los núcleos pero las aristas que entraban y salían del resto de los estados hacia los estados modificados tienen que seguir llegando y saliendo de estos.



## CONSTRUCCIÓN DE LAS TABLAS DE ANÁLISIS LALR

El análisis LALR (Lookahead - LR) es el que se utiliza normalmente en la práctica porque las tablas de análisis que genera son mucho más pequeñas que las del análisis LR canónico.

Por ejemplo, en analizador LR para un lenguaje como Pascal puede tener sobre cientos de estados.

La reducción de estados es bastante significativa.

El inconveniente es que el conjunto de gramáticas que reconoce es menor que el LR, pero con el análisis LALR se pueden expresar la mayoría de los lenguajes de programación.

Lo que se hace en el análisis LALR es buscar estados con el mismo núcleo, es decir, estados que se diferencian únicamente en los símbolos de anticipación. Y se fusionan formando un único estado.

Por ejemplo en el automata siguiente se puede ver que los estados 3 y 6 tienen el mismo núcleo.

Estado 3:

$$C \rightarrow c \circ C, c/d$$

$$C \rightarrow \bullet cC, c/d$$

$$C \rightarrow \bullet d, c/d$$

Estado 6:

$$C \rightarrow c \circ C, \$$$

$$C \rightarrow \bullet cC, \$$$

$$C \rightarrow \bullet d, \$$$

Al realizar la fusión quedaría.

$$C \rightarrow c \circ C, c/d / \$$$

$$C \rightarrow \bullet cC, c/d / \$$$

$$C \rightarrow \bullet d, c/d / \$$$

El resultado final de fusionar los estados equivalentes puede verse en la siguiente figura (Figura 2). Como puede verse los estados equivalentes eran el (3,6), (8,9) y el (4,7).

# EDUCACIÓN

SECRETARIA DE EDUCACIÓN PÚBLICA  
TECNOLÓGICO NACIONAL DE MÉXICO.

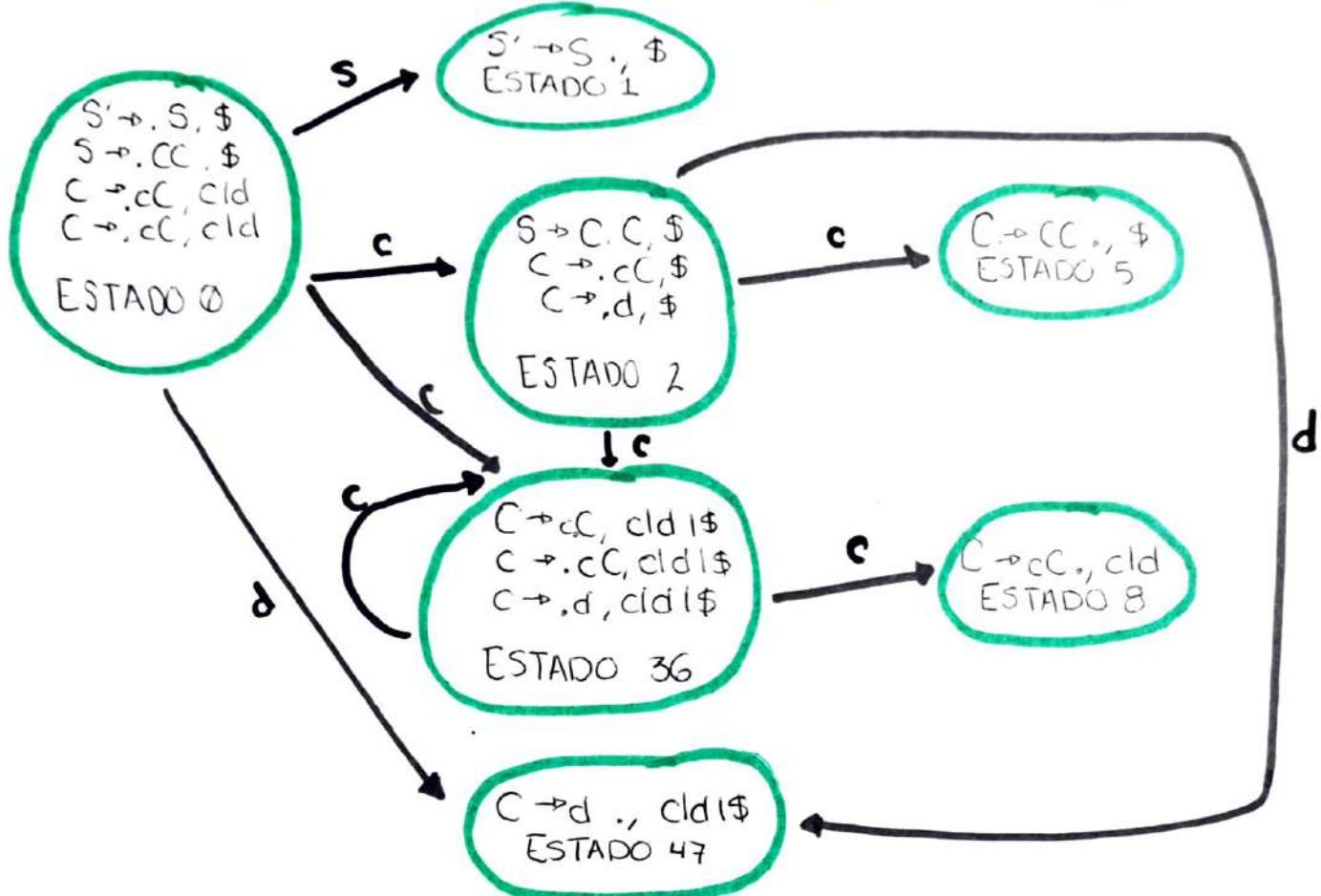


Figura 2. Autómata generado mediante el método LALR.

Con lo que ahora podríamos comprobar que la reducción del número de estados es considerable. Ahora solo queda saber las consecuencias de realizar esta fusión de estados.

Supongamos ahora que se tiene una gramática LR que no tiene conflictos de anticipación. Así que se fusionan los estados con el mismo núcleo, ahora queda resolver la cuestión de saber las consecuencias de dicha fusión. La fusión hecha puede tener un conflicto nuevo que se puede generar, es un conflicto reducción / reducción, ya que al unir dos estados puede pasar que en dos items de reducción de dos estados se tenga algún símbolo de anticipación común. Al estar separados en el análisis LR, no hay conflicto pero al juntarlos si.

Por ejemplo, en la siguiente gramática:

$$S \rightarrow aAd \mid bBd \mid aBb \mid bAe$$

$$A \rightarrow c$$

$$B \rightarrow c$$



Al hacer análisis LR no se genera ningún conflicto, pero al juntar los estados 6 y 9 se genera un conflicto nuevo reducción / reducción.

Estado 6:

$$A \rightarrow c^*, d$$

$$B \rightarrow c^*, e$$

Estado 9:

$$A \rightarrow c^*, e$$

$$B \rightarrow c^*, d$$

Nuevo estado:

$$A \rightarrow c^*, e \mid d$$

$$B \rightarrow c^*, d \mid e$$

Como puede verse hay dos items que reducen por el mismo terminal (d), con lo cual se genera un nuevo conflicto (reducción / reducción) que no existe en el análisis LR.

Este otro conflicto es el desplazamiento / reducción, pero en el análisis LALR sólo puede ocurrir si previamente el análisis LR ya lo tenía, ya que la fusión de estados tiene que ver con juntar los símbolos de anticipación y estos símbolos no tienen nada que ver con los desplazamientos.

## CONSTRUCCIÓN DE LA TABLA DE ACCIÓN / I<sub>R</sub>-A

Las tablas de construcción de análisis LALR se pueden construir directamente a partir de una gramática, pero es más fácil construirlas a partir de un conjunto de estados LR. Lo primero que se hace es fusionar los estados con el mismo núcleo, y a partir de este nuevo conjunto de estados generar de la misma forma que el análisis LR la tabla de acción y la de I<sub>R</sub>-A.

En la tabla siguiente (Tabla 2) puede verse el resultado de generar la tabla de análisis del autómata de la figura 2. Como se puede ver, el método LALR ha conseguido reducir 10 estados que generaba el LR a 7 estados que genera LALR, durante el análisis.



	C	d	\$	S	C
O	D36	D47		1	2
1			Aceptar		
2	D36	D47			5
36	D36	D47			89
47		R3	R3		
5			R1		
89	R2	R2	R2		

Tabla 2. Tabla de análisis LALR

### DESCRIPCIÓN A DETALLE DE LA CONSTRUCCIÓN DE LA TABLA DE ANÁLISIS LALR

Esta tabla se construye de la misma forma que para el método LR(0). El algoritmo de construcción consiste en realizar las siguientes acciones para cada uno de los estados:

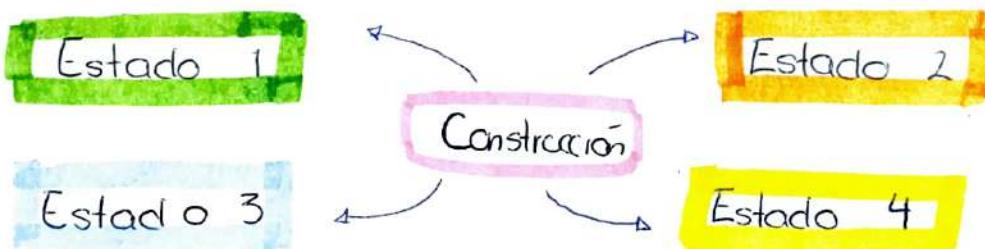


Figura 3 . Estados.

#### Estado 1

Si el estado S contiene un elemento de la forma  $[A \rightarrow aT\beta, a]$ , donde T es un terminal, se inserta en la matriz acción  $[S, T]$  la acción de desplazar al estado  $(Col S_n)$  la que va el arco etiquetado con esa terminal T y que será el que contenga como núcleo  $[A \rightarrow \alpha T, \beta a]$ .

La acción de desplazar indica que ese token se desplaza a la pila seguido del estado al que va al arco obtenido en el AFD LR(0).



## Estado 2

Si el estado S contiene un elemento de la forma  $[A \rightarrow a.N\beta, a]$  donde N es un no terminal, se inserta en la matriz  $ir-a[S, N]$  el número del estado al que va el arco etiquetado con ese no terminal ( $S_n$ ) y que será el que contenga como núcleo  $[A \rightarrow a.N\beta, a]$ .

Estas acciones en la parte de  $ir-a$  son transiciones del AFD para el no terminal correspondiente y el efecto que producen es la introducción en la cima de la pila de ese estado de transición para ese no terminal N.

## Estado 3

Si el estado S contiene un elemento completo de la forma  $[A \rightarrow a, a]$  donde a representa cualquier combinación de terminales, no terminales o cadena vacía, la acción a incluir en la matriz es reducir por la producción X, donde X es el número de la producción en la gramática, para cada uno de los elementos que haya en a.

Es decir, si los elementos en a son  $a_1$  y  $a_2$  (siendo estos terminales de la gramática o  $\$$ ), las entradas en la matriz serán acción  $[S, a_1] =$  acción  $[S, a_2] = rX$ .

## Estado 4

Si el elemento completo se refiere a la producción con la que se ha aumentado la gramática, en nuestro ejemplo  $S' \rightarrow S$ . La entrada correspondiente será acción  $[1, \$] =$  ACEPTAR.

Quedando la tabla del AFD LALR, a partir de la gramática indicada continuación, de la siguiente manera:

Gramática:

- 1:  $S' \rightarrow S$
- 2:  $S \rightarrow (L)$
- 3:  $S \rightarrow id$
- 4:  $L \rightarrow SL'$
- 5:  $L' \rightarrow SL'$
- 6:  $L' \rightarrow \lambda$

Estados	ACCIÓN					IR - A		
	id	(	)	:	\$	S	L	L'
0	d3-7	d2-6				1		
1								
2-6	d3-7	d2-6						
3-7				r3	r3	r3		
4-11				d8-13				
5						d10		
8-13				r2	r2	r2		
9				r4				
10	d3-7	d2-6						
12								
14				r5	d10			

Tabla 3. Tabla AFD LALR



F. G. M. V. A. S.

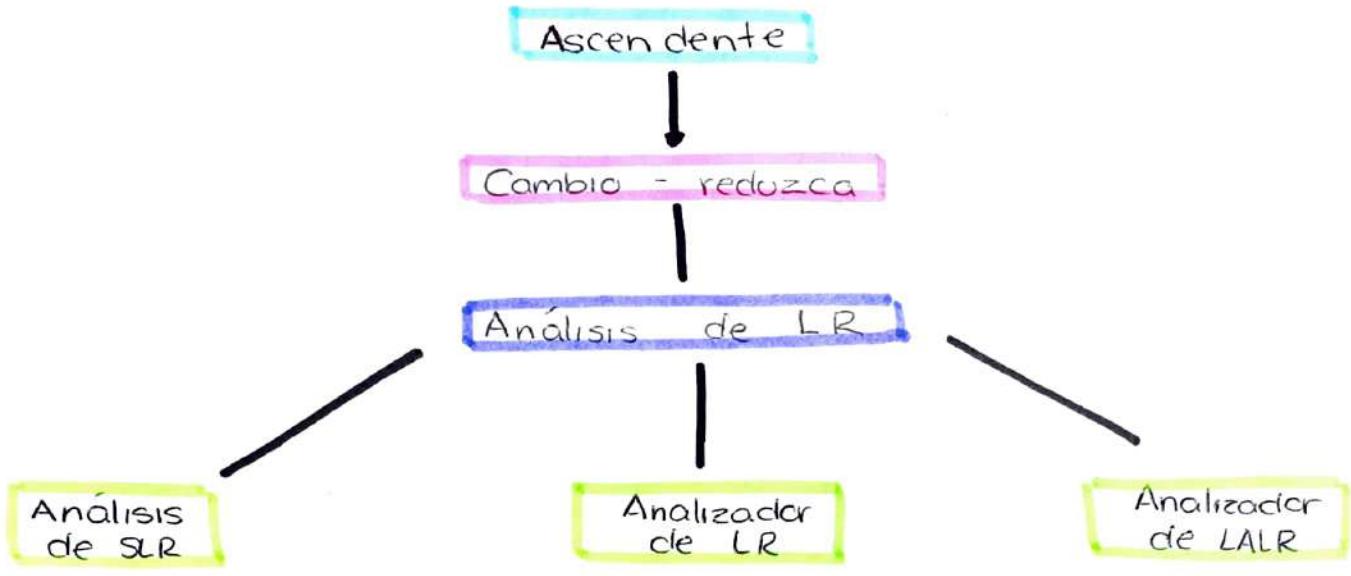


Figura 4. Compilador Diseño

## CIERRE Y CONCLUSIÓN

Los analizadores sintácticos ascendentes, particularmente los de tipo LR y sus variantes como LALR, han demostrado ser herramientas esenciales en el campo del procesamiento de lenguajes formales y la construcción de compiladores. A lo largo de este tema hemos podido explorar a lo largo de sus características, limitaciones. Tienen un gran énfasis y relevancia para el análisis sintáctico eficiente en lenguajes de programación.

El análisis LR (Left-to-Right, Rightmost derivation) se destaca por su capacidad para manejar gramáticas más complejas que los analizadores descendentes, como los LL, lo que lo convierte en una opción preferida cuando se busca mayor poder expresivo en el lenguaje a compilar. A través de técnicas de retroceso mínimo, los analizadores LR son capaces de procesar entradas de manera determinista, evitando los problemas de ambigüedad y recursividad que comúnmente afectan a los analizadores descendentes. Esto los hace robustos y eficientes, pero a la vez demandantes en cuanto a recursos computacionales, en particular en la cantidad de estados generados.

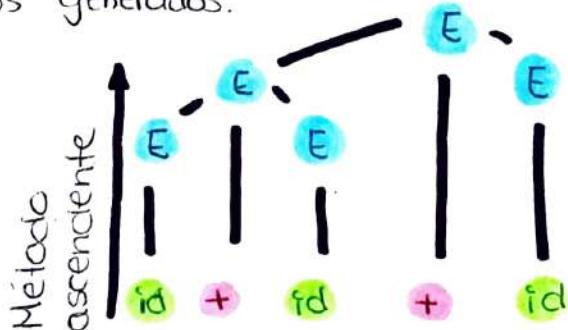


Figura 5 . Método Ascendente.



Dentro de los analizadores LR, el método SLR (Simple LR) ofrece una simplificación de análisis, aunque su capacidad para manejar ciertas gramáticas es limitada. En contraste, los algoritmos LR canónico son extremadamente poderosos, pero presentan una sobrecarga en la cantidad de estados generados. Aquí es donde entra en juego la variante LALR (Look-Ahead - LR) que combina lo mejor de ambos mundos.

La potencia de análisis LR canónico con una optimización significativa en la reducción de estados. Esto permite una implementación más ligera en cuanto a memoria, sin sacrificar la capacidad de manejar una gama amplia de gramáticas contextuales.

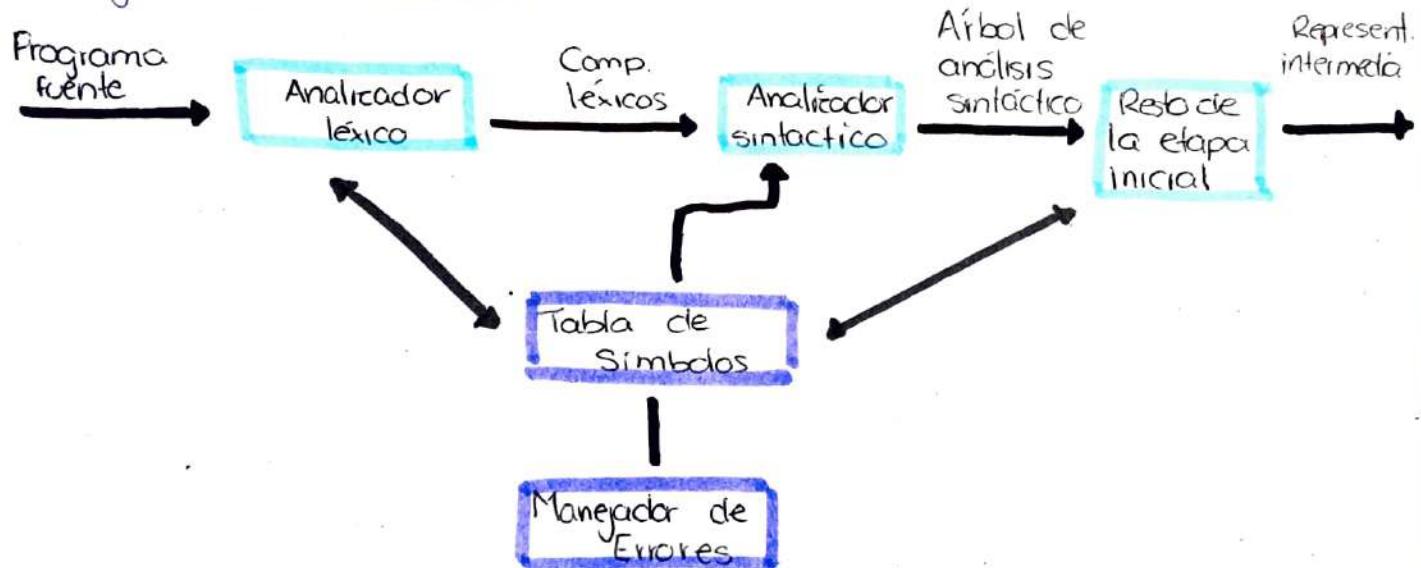


Figura 6. Tipos de Analizador sintáctico.

Otro punto relevante es que las técnicas de análisis LR y LALR han permitido la creación de herramientas de desarrollo más eficientes, como generadores automáticos de analizadores, que simplifican el trabajo de los desarrolladores al construir compiladores para lenguajes específicos. Esto ha resultado en una aceleración de desarrollo, permitiendo que nuevos lenguajes y herramientas lleguen más rápido al mercado.

En conclusión, los analizadores sintáticos ascendentes LR y LALR representan un avance significativo en el campo del análisis de los lenguajes formales. Su capacidad para manejar gramáticas complejas, combinada con su adaptabilidad y optimización, los convierte en una herramienta clave no solo en la construcción de compiladores, sino en el diseño de nuevos lenguajes de programación. A medida que los lenguajes continúan evolucionando, es probable que estos métodos también se adopten y mejoren.

# DISEÑO Y ADMINISTRACIÓN DE UNA TABLA DE SÍMBOLOS.

## INTRODUCCIÓN

La tabla de símbolos es una estructura de datos fundamental en el diseño de compiladores y ensambladores, ya que permite almacenar información crucial sobre los identificadores usados en un programa, como variables, constantes, funciones, clases y sus atributos. El propósito principal de una tabla de símbolos es gestionar toda la información relacionada con los nombres y las entidades que esos nombres representan durante el proceso de traducción del código fuente, permitiendo que el compilador verifique la correcta utilización de los elementos dentro de su ámbito. El correcto diseño y administración de esta tabla es esencial para garantizar la eficiencia del compilador y la corrección del programa resultante.

## FUNCIones PRINCIPALES DE LA TABLA DE SÍMBOLOS

1- Almacenamiento de información de los identificadores: La tabla de símbolos almacena información sobre cada identificador del programa, que incluye el nombre del identificador, su tipo de dato, ámbito o contexto en el que está definido, la dirección de memoria (si aplica) y otras propiedades relevantes.

2- Verificación Semántica: Durante el análisis semántico, el compilador consulta la tabla de símbolos para asegurarse de que los identificadores se utilicen de manera coherente con su declaración. Por ejemplo, verificar que una variable ha sido declarada antes de ser utilizada o que las operaciones se realicen entre tipos compatibles.

3- Manejo de ámbitos: La tabla de símbolos debe de ser capaz de gestionar los diferentes ámbitos (scopes) de un programa, que pueden ser globales o locales, especialmente en lenguajes que permiten el uso de funciones o bloques anidados.

4- Optimización: En fases posteriores, como la generación de código y la optimización, la tabla de símbolos puede ayudar a reducir el consumo de recursos, eliminando variables que no se utilizan o reorganizando el uso de recursos y registros.

## DISEÑO DE LA TABLA DE SÍMBOLOS.

El diseño de una tabla de símbolos debe tener en cuenta factores como la estructura de datos utilizada, el tipo de información que debe almacenar y cómo se gestionan los diferentes ámbitos del programa.

### 1. ESTRUCTURA DE DATOS

Existen varias estructuras de datos que pueden usarse para implementar tablas de símbolos, cada una con sus ventajas y desventajas:

- Tablas Hash: Una de las estructuras más comunes para implementar tablas de símbolos. El identificador se mapea a una clave a través de una función hash, lo que permite una búsqueda eficiente en tiempo constante. Es útil para programas grandes con muchos identificadores.

- Árboles Balanceados (AVL o RB-trees): Son útiles en situaciones donde se necesita un orden lexicográfico de los identificadores. El costo de inserción, búsqueda y eliminación es algorítmica lo que garantiza una buena eficiencia.

- Listas Enlazadas: Son fáciles de implementar y manejar en lenguajes simples o programas pequeños, pero su rendimiento decrece con el aumento de número de identificadores, ya que las búsquedas tienen un costo lineal.

## 2. INFORMACIÓN ALMACENADA.

La tabla de simbolos puede almacenar una amplia variedad de información sobre los identificadores, dependiendo del lenguaje de programación.

La información básica generalmente incluye:

- Nombre Del Identificador: El nombre que aparece en el código fuente.
- Tipo De Dato: El tipo de dato (entero, flotante, Cadena, etc) de una variable o el tipo de retorno de una función.
- Ámbito: El contexto en el que está definido el identificador (global, local, dentro de una función, etc).
- Dirección De Memoria: La ubicación en memoria donde está almacenado el valor del identificador (si aplica).
- Tamaño Y Dimensión: Para identificadores que son arreglos o estructuras, es necesario almacenar su tamaño y dimensión.
- Parámetros: En el caso de funciones, la tabla de simbolos puede almacenar información sobre los parámetros formales.

## 2. INFORMACIÓN ALMACENADA.

La tabla de símbolos puede almacenar una amplia variedad de información sobre los identificadores, dependiendo del lenguaje de programación. La información básica generalmente incluye:

- Nombre Del Identificador: El nombre que aparece en el código fuente.
- Tipo De Dato: El tipo de dato (entero, flotante, Cadena, etc) de una variable o el tipo de retorno de una función.
- Ámbito: El contexto en el que está definido el identificador (global, local, dentro de una función, etc).
- Dirección De Memoria: La ubicación en memoria donde está almacenado el valor del identificador (si aplica).
- Tamaño Y Dimensión: Para identificadores que son arreglos o estructuras, es necesario almacenar su tamaño y dimensión.
- Parámetros: En el caso de funciones, la tabla de símbolos puede almacenar información sobre los parámetros formales.

### 3. MANEJO DE ÁMBITOS

El manejo de ámbitos (scoping) es fundamental para que un compilador pueda determinar correctamente donde y cómo se deben utilizar los identificadores. Un programa puede tener múltiples ámbitos jerárquicos, como funciones, bucles o bloques. Los ámbitos permiten que diferentes partes de un programa puedan utilizar los mismos nombres de identificadores sin causar conflictos.

Existen varias estrategias para gestionar los ámbitos:

- Árbol de tablas de Símbolos: Cada nuevo ámbito se representa como un nodo hijo del ámbito actual. Esto permite que el compilador pueda buscar identificadores en el ámbito actual y, si no se encuentran, moverse hacia ámbitos más amplios o globales.

- Pilas de tablas de símbolos: Otra técnica común es usar una pila. Cada vez que se entra en un nuevo ámbito, se crea una nueva tabla de símbolos y se apila. Al salir del ámbito, se desapila la tabla de símbolos correspondiente.

Ejemplo de Manejo de Ámbitos:

```
int x; // Declaración en el ámbito global
```

```
void f() {
```

```
    int x; // Nueva declaración en el ámbito local
```

```
}
```

```
    int x; // Declaración en un subámbito local
```

```
}
```

En este ejemplo, la tabla de símbolos debe ser capaz de distinguir entre los tres diferentes x en función de su ámbito.

## ADMINISTRACIÓN DE LA TABLA DE SÍMBOLOS

La administración eficiente de una tabla de símbolos implica asegurar un correcto balance entre la inserción, búsqueda y eliminación de identificadores, además de la gestión de memoria. Algunos factores a considerar son:

### I: Inserción De Identificadores

Cada vez que se encuentra un identificador en el análisis sintáctico, se debe insertar en la tabla de símbolos con la información relevante. Para evitar duplicados, se verifica si el identificador ya ha sido declarado en el mismo ámbito antes de realizar la inserción.

Ejemplo:

Si el código tiene:

```
int x = 5;
```

```
int x = 10; //Error: Redeclaración en el mismo  
              //ámbito
```

El compilador debería generar un error durante la inserción, ya que x ya está definido en el mismo ámbito.

## 2- Busqueda De Identificadores.

La búsqueda es una operación frecuente, ya que el compilador necesita acceder a la información del identificador durante la fase de análisis semántico. Una tabla de símbolos bien diseñada debe permitir una búsqueda rápida. Si se utiliza una estructura jerárquica de ámbitos, la búsqueda comienza en el ámbito actual y luego sube en la jerarquía de ámbitos superiores hasta llegar al ámbito global.

Ejemplo:  
En el siguiente código:

```
int x = 5;
```

```
Void f() {
```

```
    printf("%d", x); //Aquí el compilador busca "x" primero  
                      //en el ámbito local, luego en el  
                      //global.
```

Como x no se encuentra en el ámbito local, el compilador buscará en el ámbito global.

### 3- Eliminación De Identificadores

Cuando el compilador sale de un ámbito, debe eliminar los identificadores asociados a ese ámbito para liberar memoria y evitar conflictos en futuros análisis. Este proceso es simple cuando se utilizan estructuras como pilas, ya que basta con desapilar la tabla de símbolos correspondiente al ámbito que se abandona.

### 4- Optimización Del Uso De La Memoria

En lenguajes de programación con un gran número de identificadores y ámbitos, es importante optimizar el uso de la memoria para evitar el agotamiento de recursos. El diseño de la tabla de símbolos debe estar orientado a un uso eficiente de la memoria, liberando las entradas de la tabla que ya no se necesitan cuando los identificadores salen de su ámbito de validez.

Ejemplo de una tabla de símbolos.

Consideremos un fragmento de código en C:

```
int a=10;  
void f() {  
    int b=20;  
    {  
        int c=30;  
    }  
}
```

En este caso, la tabla de símbolos podría representarse como:

Identificador	Tipo	Ambito	Dir. Memoria	Valor
a	int	global	0x1001	10
b	int	f()	0x1002	20
c	int	bloque	0x1003	30

### Conclusión

El diseño y administración de la tabla de símbolos es un aspecto fundamental en el desarrollo de compiladores. Su implementación eficiente garantiza una correcta verificación semántica, así como una gestión adecuada de los identificadores y sus respectivos ámbitos. Además, una estructura de datos bien seleccionada para la tabla de símbolos puede optimizar las operaciones de búsqueda, inserción y eliminación, mejorando así el rendimiento del compilador. Una buena administración también asegura el correcto manejo de la memoria, sobre todo en programas extensos.

# MANEJO DE ERRORES SINTÁCTICOS Y SU RECUPERACIÓN.

## INTRODUCCIÓN

El análisis sintáctico es una fase crucial en la construcción de compiladores, donde se verifica si la estructura de una secuencia de tokens, generada por el análisis léxico, sigue las reglas de una gramática predefinida. Sin embargo, durante el análisis sintáctico, es común que ocurran errores debido a secuencias no válidas en el código fuente. Estos errores son conocidos como errores sintácticos y pueden resultar en programas incorrectos. El manejo de errores sintácticos y su recuperación son procesos que buscan no solo identificar los errores, sino también continuar el análisis de manera que se puedan detectar más errores y proporcionar retroalimentación significativa al programador.

## TIPOS DE ERRORES SINTÁCTICOS

1- Errores por omisión: Se producen cuando se omite algún elemento esencial de la gramática, como un parentesis de cierre o un punto y coma, ejemplo:

int x = 10 // Falta el punto y coma.

2- Errores por inclusión: Estos ocurren cuando se introduce un símbolo innecesario o incorrecto en la secuencia de entrada, ejemplo:

int x = 10;; // punto y coma adicional.

3- Errores de Orden: Aparecen cuando los elementos están en un orden incorrecto. Por ejemplo un operador antes de una expresión:

x = \* 5; // operador antes de la expresión.

4- Errores por estructuras mal anidadadas: Se dan cuando los elementos estructurales como parentesis o llaves no están bien balanceados. Ejemplo:

if (x > 10 { // llave de apertura sin parentesis de cierre.

## ESTRATEGIAS DE MANEJO DE ERRORES SINTÁCTICOS

El manejo de errores sintácticos implica dos tareas principales: detección de errores y recuperación. El objetivo es identificar el punto en el que ocurre el error y, si es posible, continuar el análisis.

### 1- Detección temprana de errores

La detección temprana de errores es un principio fundamental para garantizar que un compilador ofrezca una retroalimentación oportuna. Los analizadores sintácticos detectan errores en el momento en que encuentran una secuencia de símbolos que no sigue las reglas de la gramática.

Ejemplo: Si el analizador espera un operador entre dos expresiones, pero encuentra dos identificadores consecutivos, esto indica un error.

### 2 Mecanismos de recuperación de errores

El siguiente paso es intentar recuperar el análisis y continuar procesando el resto del código. Esto permite encontrar más errores en el programa y no solo el primer error que interrumpió el análisis.

Existen varias técnicas de recuperación que se emplean comúnmente:

## a. Recuperación por pánico

La recuperación por pánico es una técnica simple que se utiliza en situaciones donde el error puede ser "saltado" de manera segura para continuar con el análisis. El método consiste en descartar tokens hasta encontrar un delimitador seguro, como un punto y coma, que generalmente indica el final de una instrucción. Esto permite al compilador seguir analizando el resto del código.

### Ventajas:

- Fácil de implementar
- Asegura que el analizador no se quede atascado en un bucle infinito.

### Desventajas

- Puede omitir una gran parte del código, perdiendo la oportunidad de detectar otros errores sintácticos.

### Ejemplo:

```
int x = 10  
int fx = 20;
```

El punto y coma faltante en la primera línea provocará un error. El analizador entrará en modo pánico y saltará tokens hasta encontrar ; en la segunda línea, permitiendo que el análisis continúe desde ahí.

## b. Recuperación basada en inserciones y eliminaciones.

Este enfoque implica modificar el código para hacer que siga las reglas de la gramática, insertando o eliminando tokens. Esto es especialmente útil cuando los errores son simples y parecen ser omisiones o inclusiones accidentales.

### Ventajas:

- Permite una recuperación más precisa, corrigiendo pequeños errores.
- Puede ser más amigable para el usuario, ya que el analizador sugiere posibles correcciones.

### Desventajas:

- Requiere una lógica más compleja para decidir qué tokens insertar o eliminar.
- Podría generar nuevas ambigüedades si no se manejan con cuidado.

### Ejemplo:

int x = 10 //Falta un punto y coma.

El analizador puede sugerir la inserción de un ; al final de la declaración.

### c. Recuperación mediante análisis local.

En este enfoque, el analizador intenta corregir el error basándose en el contexto inmediato del error, modificando una pequeña parte del código. Esto puede implicar la inserción de un token que falta, o la corrección de un símbolo inesperado.

#### Ventajas

- Proporciona una corrección específica en el lugar del error
- Permite que el análisis continúe sin saltar una gran cantidad de código.

#### Desventajas

- Puede generar correcciones incorrectas si no se aplica con cuidado.
- Requiere un análisis más detallado para determinar la corrección adecuada.

### d. Recuperación mediante backtracking

El backtracking es una técnica en la que el analizador, tras encontrar un error, retrocede hasta un punto anterior en el análisis e intenta encontrar una ruta alternativa que no genere el error. Esta técnica es más común en analizadores descendentes y puede ser computacionalmente costosa.

## Ventajas:

- Es capaz de explorar múltiples opciones antes de reportar un error.
- Puede encontrar caminos válidos en caso de ambigüedad

## Desventajas:

- Puede ser inefficiente, ya que requiere retroceder y volver a analizar la entrada.
- No siempre garantiza una solución correcta y puede entrar en ciclos infinitos.

Ejemplo práctico de manejo de errores

Consideremos el sig. código en C:

```
int main() {  
    int x=10  
    printf ("Valor de x: %d", x);  
}
```

Aquí el analizador detectará que falta un punto y coma después de "int x=10". Dependiendo de la estrategia utilizada:

- Recuperación por pánico: El analizador podría saltar hasta encontrar el delimitador ; en la línea siguiente.
- Inserción de tokens: El compilador podría sugerir insertar un ";" justo después de "int x=10"
- Backtracking: El analizador podría retroceder e intentar reinterpretar la linea anterior, aunque es menos eficiente en este caso.

## HERRAMIENTAS PARA LA RECUPERACION DE ERRORES

Existen diversas herramientas de desarrollo que implementan técnicas de manejo de errores sintácticos avanzados, tales como Bison y ANTLR, las cuales permiten definir gramáticas y especificar mecanismos para el manejo y la recuperación de errores.

### Conclusión.

El manejo de errores sintácticos es una parte integral en el diseño de compiladores y herramientas de análisis de lenguajes de programación. La capacidad de detectar y recuperar errores de manera efectiva no solo mejora la calidad del código generado, sino que también ofrece una mejor experiencia al desarrollador, permitiéndole corregir errores más rápidamente. Las técnicas de recuperación, como la recuperación por pánico, las inserciones y eliminaciones, y el análisis local, proporcionan enfoques diversos según el tipo de error y las necesidades del compilador.

## Generadores de Código para Analizadores Sintácticos: Yacc y Bison

Los llamados generadores de código son herramientas que ayudan a los desarrolladores en la creación de analizadores sintácticos mediante especificaciones de gramáticas formales.

Estas herramientas generan código en lenguajes de programación, como C, C++ y Java, aunque en el caso de Yacc solo se genera el código en lenguaje C, mientras que en Bison el código se puede generar tanto en C, C++, Java y más.

Este código implementa el analizador sintáctico de acuerdo a la gramática o especificada.

Las dos herramientas más conocidas para este propósito son Yacc y Bison.

### <Yacc (Yet Another Compiler Compiler)>

Desarrollado en los 70 y formando parte del sistema UNIS, fue uno de los primeros generadores de analizadores sintácticos. Los analizadores sintácticos que genera están basados en gramáticas LALR (Look-Ahead LR).

Ademas, hace uso de una notación específica para poder describir la gramática del lenguaje y en consecuencia producir un programa en C capaz de reconocer y analizar el código fuente según la estructuras definidas en la gramática.

### Gramáticas BNF

La notación que usan Yacc para definir las gramáticas del lenguaje es la Backus - Norr Form (BNF). Ademas hace uso de un archivo de entrada en el cual se describe la especificación gramatical que posteriormente sera utilizado en la generación del código para el analizador sintáctico.

### Arboles de análisis

Un resultado típico dado por un analizador Yacc es un árbol de análisis, este árbol representar la estructura jerárquica del código fuente segun la gramática dada.

## Especificación

YACC



Funciónde análisis  
sintáctico

y.tab.c (BYACC)

y.tab.c (UNIX)

## [Bison]

Derivado de YACC y desarrollado por el proyecto GNU, fue propuesto y diseñado para mantener compatibilidad con YACC, dando como resultado funcionalidades similares, pero dado a su modernidad y mejoras respecto a YACC, es elegido por sobre este.

## Ventajas sobre YACC

Bison es multilingüe lo cuál lo hace capaz de generar analizadores en C++ o Java, facilitando la integración con proyectos orientados a objetos.

También incluye una mejor estrategia de recuperación de errores, permitiendo especificar reglas que continúen el análisis sintáctico aunque se encuentren errores.

Al formar parte del proyecto GNU, Bison es usado en proyectos de software libre y con soporte para múltiples plataformas.

Bison también provee de herramientas adicionales para personalizar el comportamiento del analizador, como pasar la ejecución de un AST (Árbol de sintaxis abstracta) y soporte de gramáticas más complejas.

### Comparación entre YACC y Bison

Característica	YACC	Bison
Lenguajes Soportados	C	C, C++, Java
Portabilidad	Específica para UNIX	Multiplataforma
Compatibilidad	Lex	Lex/Flex
Extensibilidad	Limitada	Soporte de gramáticas complejas
Popularidad	Antiguo, ampliamente utilizado	Amplio uso en proyectos modernos

## [Conclusion]

YACC y Bison son herramientas esenciales en el desarrollo de compiladores, analizadores sintácticos y herramientas de procesamiento de lenguajes. Aunque YACC siga siendo una herramienta poderosa yiable, Bison ha ganado popularidad por su flexibilidad, mejor manejo de errores y soporte para múltiples lenguajes de programación.

## Referencias

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, techniques, and tools (2nd ed.). Pearson Education.

Muchnick, S. S. (1997). Advanced compiler design and implementation. Morgan Kaufmann.

Hunter, R. (2006). Advanced error recovery techniques for parser design. ACM Computing Surveys, 38(3), 1-32.

Bibliografía de autómatas y lenguajes formales Alma María Pisabarro Marrón, T. (s/f). El generador de analizadores sintácticos Yacc IV. Uva.es. Recuperado el 24 de septiembre de 2024, de <https://www.infor.uva.es/~mluisa/talf/docs/lab0/L11.pdf>

SOFTWARE DE GENERACIÓN Y SIMULACIÓN DE TABLAS DE ANÁLISIS SINTÁCTICO (BURGRAM) - Construcción de las tablas de análisis LALR. (s/f). Cgosorio.es. Recuperado el 24 de septiembre de 2024, de [http://cgosorio.es/BURGRAM/indexac26.html?Los\\_algoritmos\\_de\\_an%C3%A1lisis%0Asint%C3%A1ctico%0AAnal%C3%A1sis\\_Sint%C3%A1ctico%0AAAscendente:Construcci%C3%B3n\\_de\\_las\\_tablas\\_de%0Aan%C3%A1lisis\\_LALR](http://cgosorio.es/BURGRAM/indexac26.html?Los_algoritmos_de_an%C3%A1lisis%0Asint%C3%A1ctico%0AAnal%C3%A1sis_Sint%C3%A1ctico%0AAAscendente:Construcci%C3%B3n_de_las_tablas_de%0Aan%C3%A1lisis_LALR)

Tipos de Analizador Sintáctico. (2016, junio 2). Eportafolio compiladores UGB.

<https://compiladoresugb.wordpress.com/2016/06/01/tipos-de-analizador-sintactico/>

Universidad de Guanajuato. (2022, julio 16). Clase digital 4. Herramientas de software: Yacc/Bison. Recursos Educativos Abiertos; Sistema Universitario de Multimodalidad Educativo (SUME) - Universidad de Guanajuato. <https://blogs.ugto.mx/rea/clase-digital-4-herramientas-de-software-yacc-bison/>

(S/f-a). Cartagena99.com. Recuperado el 24 de septiembre de 2024, de [https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2\\_M4\\_U3\\_T4.pdf](https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2_M4_U3_T4.pdf)

(S/f-b). Ecured.cu. Recuperado el 24 de septiembre de 2024, de [https://www.ecured.cu/Analizador\\_sint%C3%A1ctico\\_LL](https://www.ecured.cu/Analizador_sint%C3%A1ctico_LL)

(S/f-c). Uhu.es. Recuperado el 24 de septiembre de 2024, de [https://www.uhu.es/francisco.moreno/gii\\_pl/docs/Tema\\_3.pdf](https://www.uhu.es/francisco.moreno/gii_pl/docs/Tema_3.pdf)

Llamas, L. (2023, junio 1). Precedencia de operadores. Luis Llamas. <https://www.luisllamas.es/programacion-precedencia-operadores/>

Sintaxis Spread. (s/f). MDN Web Docs. Recuperado el 24 de septiembre de 2024, de [https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Spread\\_syntax](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Spread_syntax)

Rodríguez, J. M., & Pérez, J. F. (2008, enero 23). Generador de analizadores sintácticos BISON. CCIA, Universidad de Vigo. <https://ccia.esei.uvigo.es/docencia/PL/bison.pdf>

AIX 7.3. (2023, March 24). <https://www.ibm.com/docs/es/aix/7.3?topic=information-creating-parser-yacc-program>

CARRERA	NOMBRE DE LA ASIGNATURA
INGENIERIA EN SISTEMAS COMPUTACIONALES	LENGUAJES Y AUTOMATAS 2

DOCENTE DESIGNADO
ISC. RICARDO GONZÁLEZ GONZÁLEZ

PRACTICA No.	NOMBRE DE LA PRACTICA	DURACIÓN (HORAS)
3	¿CÓMO FUNCIONA UN ANALIZADOR DESCENDENTE?	3 horas

1	INTRODUCCIÓN
	Un analizador descendente es un tipo de parser que analiza una cadena de entrada de arriba hacia abajo, comenzando desde el símbolo inicial de la gramática y tratando de derivar la cadena completa siguiendo las reglas de producción. El proceso involucra descomponer la cadena de entrada en fragmentos más pequeños, tratando de hacer coincidir los fragmentos con las producciones de la gramática.
	<b>Principios básicos:</b>
	Simbolización del análisis: Un analizador descendente trata de construir un árbol de derivación desde el símbolo inicial hasta las hojas (terminales) que coinciden con la entrada.
	Funciones recursivas: En los analizadores descendentes recursivos, se usa una función recursiva para cada no terminal. Esta función trata de expandir las producciones de ese no terminal, emparejando los símbolos terminales de la cadena.
	<b>Estrategias comunes:</b>
	Análisis descendente recursivo: Implementa el análisis a través de llamadas recursivas, donde cada función representa una regla de la gramática.
	Análisis predictivo (LL): Usa la técnica LL(k) (el primer "L" indica que lee la entrada de izquierda a derecha, y el segundo "L" que produce una derivación por la izquierda). Este método se basa en predecir qué regla de producción usar, mirando los próximos tokens de la entrada. Se evita el backtracking (retroceso), ya que se sabe qué producción elegir.
	Conflictos PRIMERO/PRIMERO: Un problema común es que múltiples producciones pueden comenzar con el mismo símbolo, lo que genera incertidumbre en cuál regla aplicar, y por eso es necesario resolver estos conflictos.
	<b>Aplicaciones:</b> Los analizadores descendentes son comunes en compiladores e intérpretes, ya que

permiten descomponer el código fuente en estructuras lógicas que pueden ser interpretadas o ejecutadas.

Este tipo de analizador es más fácil de implementar, pero puede tener limitaciones si la gramática es muy compleja o ambigua.

**2**

### **OBJETIVO ( COMPETENCIAS )**

- Comprender el funcionamiento de un analizador descendente.
- Ser capaz de implementar un analizador descendente en un lenguaje de programación.
- Identificar los problemas más comunes en el análisis descendente, como los conflictos PRIMERO/PRIMERO.

**3**

### **MARCO TEÓRICO REFERENCIAL**

#### **✓ Gramáticas libres de contexto**

Las gramáticas libres de contexto (GLC) son un tipo de gramática formal que se utilizan para definir lenguajes en la teoría de autómatas y lenguajes formales. Son especialmente útiles para describir la estructura de lenguajes de programación y lenguajes naturales.

Características principales:

Producciones: Las reglas de producción en una GLC tienen la forma  $A \rightarrow \alpha$ , donde:

$A$

$A$  es un símbolo no terminal (como una variable que representa una estructura en el lenguaje).

$\alpha$

$\alpha$  es una secuencia de símbolos, que pueden ser terminales (símbolos finales, como letras o palabras) o no terminales.

Libres de contexto: Esto significa que el lado izquierdo de cada regla de producción siempre es un solo símbolo no terminal, sin importar el contexto en el que aparezca en la cadena. A diferencia de las gramáticas sensibles al contexto, las GLC no dependen del entorno de un símbolo para aplicar las reglas.

Generación de lenguajes: Las GLC generan lenguajes libres de contexto, que son un subconjunto de los lenguajes formales. Este tipo de lenguajes puede ser reconocido por un autómata de pila (PDA, Pushdown Automaton).

Ejemplo:

**Ejemplo:**

Considera la gramática  $G$  con las siguientes producciones:

- $S \rightarrow aSb$
- $S \rightarrow \epsilon$

Esta gramática puede generar cadenas del lenguaje  $L = \{a^n b^n \mid n \geq 0\}$ , como  $ab, aabb, aaabbb$ , etc.

Las GLC son fundamentales para construir analizadores sintácticos (parsers), que son utilizados en compiladores para validar la estructura de código fuente.

✓ **Algoritmos de análisis descendente**

Los algoritmos de análisis descendente son técnicas utilizadas para analizar una cadena de entrada (como una secuencia de código fuente) y determinar si pertenece a un lenguaje definido por una gramática libre de contexto. Estos algoritmos siguen el enfoque de "arriba hacia abajo", comenzando desde el símbolo inicial de la gramática y tratando de construir un árbol de derivación que genere la cadena de entrada.

**Tipos de algoritmos de análisis descendente:**

**1. Análisis descendente recursivo:**

- Utiliza llamadas recursivas a funciones que corresponden a las reglas de la gramática.
- Intenta derivar la cadena de entrada aplicando las reglas de producción de manera recursiva.
- Es sencillo de implementar, pero no siempre es eficiente, especialmente cuando la gramática tiene recursión por la izquierda.

**Ejemplo:** Si tienes una gramática con la producción:

- $S \rightarrow aSb$  El algoritmo trataría de emparejar un **a**, luego llamaría a sí mismo para emparejar la parte **s**, y luego intentaría emparejar un **b**.

**2. Análisis descendente con retroceso (backtracking):**

- Prueba todas las posibles producciones de una gramática para ver cuál genera la cadena de entrada.
- Si una opción falla, retrocede y prueba otra.
- No es muy eficiente, ya que puede implicar muchos retrocesos y reintentos.

**3. Análisis descendente predictivo:**

- Es una mejora del análisis descendente con retroceso, que evita la necesidad de retroceder usando una tabla de predicción.
- Se basa en gramáticas que son LL(1), lo que significa que para cualquier símbolo no terminal y símbolo terminal de la cadena de entrada, la próxima regla de producción a aplicar puede decidirse viendo solo el siguiente símbolo de la entrada (es decir, con una mirada hacia adelante de un símbolo).
- El análisis predictivo usa una tabla de análisis que guía el proceso, haciendo que sea mucho más eficiente que el análisis con retroceso.

**Proceso general:**

- 1.Iniciar desde el símbolo inicial de la gramática.
- 2.Intentar derivar la cadena de entrada aplicando reglas de producción de la gramática.
- 3.Leer símbolos de la cadena de entrada y tratar de hacer coincidir con las producciones.
- 4.Si una derivación falla, puede intentarse otra (si se usa retroceso) o detectar el error de manera inmediata (en el caso del análisis predictivo).

**Ejemplo de análisis predictivo:**

Para la gramática:

- $S \rightarrow aS \mid b$

Y la cadena de entrada **aab**, el algoritmo:

1. Comienza con **S**.
2. Al ver el primer **a**, aplica la regla  $S \rightarrow aS$ .
3. Para el segundo símbolo **a**, aplica nuevamente  $S \rightarrow aS$ .
4. Finalmente, cuando se encuentra con **b**, aplica  $S \rightarrow b$ , derivando correctamente la cadena.

**Ventajas:**

- Deterministas: En el caso de gramáticas LL(1), el análisis predictivo no necesita retroceder ni reintentar opciones.
- Implementación sencilla: El análisis descendente recursivo es intuitivo y fácil de implementar para pequeñas gramáticas.

**Desventajas:**

- Limitaciones con gramáticas: No todas las gramáticas libres de contexto son adecuadas para análisis descendente (por ejemplo, aquellas con recursión por la izquierda).
- No es tan eficiente: El retroceso puede ser costoso en términos de tiempo de ejecución en gramáticas grandes.

El análisis descendente es clave en la construcción de analizadores sintácticos para compiladores, que validan la estructura de programas escritos en lenguajes de programación.

✓ **Conflictos PRIMERO/PRIMERO.**

Los conflictos PRIMERO/PRIMERO ocurren en gramáticas utilizadas por analizadores sintácticos descendentes predictivos, especialmente en los que utilizan la técnica LL(1), que es un tipo de parser que toma una decisión basándose en un token de entrada y en la producción correspondiente.

Estos conflictos se presentan cuando, al intentar derivar una producción para una regla gramatical, el conjunto PRIMERO de dos o más alternativas posibles coincide. En otras palabras, cuando hay dos producciones para un mismo no terminal, y al revisar el siguiente símbolo de entrada no es posible decidir cuál producción usar, se genera el conflicto.

Por ejemplo, en la siguiente gramática:

$$A \rightarrow \alpha \mid \beta$$

Si los conjuntos PRIMERO de ` $\alpha$ ` y ` $\beta$ ` tienen algún símbolo en común, el analizador no podrá decidir cuál producción utilizar.

Ejemplo concreto:

$$A \rightarrow aX \mid aY$$

Si ambos ` $aX$ ` y ` $aY$ ` tienen el símbolo terminal `a` en sus conjuntos PRIMERO, el analizador LL(1) no sabrá cuál de las dos reglas elegir, lo que provoca un conflicto PRIMERO/PRIMERO.

Para resolver este tipo de conflictos, se pueden aplicar varias técnicas, como eliminación de la recursión por la izquierda o *factorización de la gramática*.

4

#### MATERIALES UTILIZADO

- Computadora.
- Compilador o entorno de programación (**Java**).
- Una gramática libre de contexto que usarás para construir el analizador.

5

#### REQUISITOS BÁSICOS.

- Conocimiento sobre gramáticas y autómatas.
- Familiaridad con un lenguaje de programación.
- Entender los conceptos de análisis léxico y análisis sintáctico.

6

#### DESARROLLO DE LA PRÁCTICA (PASO A PASO)

##### Ejemplo y su explicación:

Implementaremos el ejemplo de un analizador sintáctico recursivo descendente para una gramática simple en Java. Está diseñado para verificar si una cadena de entrada sigue una gramática muy básica, que corresponde a expresiones aritméticas formadas por "números" (representados por el símbolo num) y el operador +.

### Desglose de lo que hace el código:

#### Entrada y estado:

La cadena de entrada es "num+num", donde se supone que "num" representa un número. index se usa para mantener el seguimiento del token actual que se está analizando en la cadena. Reglas de la gramática: La gramática que sigue el código es la siguiente:

$$\begin{aligned}E &\rightarrow TE' \\E' &\rightarrow + TE' \mid \epsilon \\T &\rightarrow \text{num}\end{aligned}$$

Esta gramática describe expresiones aritméticas que consisten en uno o más términos (num) separados por el operador +.

#### Funciones recursivas:

E(): Corresponde a la regla  $E \rightarrow TE'$ . Primero, verifica si hay un término ( $T()$ ), y luego intenta analizar el resto de la expresión ( $E'$ ).

Ep(): Implementa la regla  $E' \rightarrow + TE' \mid \epsilon$ . Si encuentra un +, espera que siga otro término ( $T()$ ) y luego recursivamente verifica si hay más operadores + (llamando de nuevo a Ep()). Si no hay +, se asume que se llegó al final de la expresión (regla  $E' \rightarrow \epsilon$ ).

T(): Corresponde a la regla  $T \rightarrow \text{num}$ . Verifica si el token actual es un número (n en este caso simula un número).

#### Análisis de la cadena:

El programa comienza llamando a E(), la función que representa la expresión completa. Si esta función retorna true y todos los tokens en la cadena han sido procesados, la cadena es válida según la gramática. Si alguna de las funciones no puede validar la parte correspondiente de la gramática, la cadena es inválida.

Salida:

Si la cadena es válida según la gramática, se imprime "Cadena válida según la gramática.". Si no lo es, se imprime "Cadena inválida.".

Ejemplo:

Dada la cadena "num+num", el analizador hará lo siguiente:

Llama a E().

Dentro de E(), llama a T() para analizar el primer num, que es válido.

Luego, llama a Ep() para ver si hay un +. Encuentra el +, por lo que sigue analizando el siguiente num llamando de nuevo a T(), que también es válido.

Al no haber más operadores +, Ep() retorna true, y como toda la cadena fue procesada, el programa concluye que la cadena es válida.

En resumen, este código valida cadenas que siguen la forma num+num+num..., donde puede haber cualquier cantidad de términos num separados por +.

## Código en java:

```
public class Main {
    static String input;
    static int index = 0;

    public static void main(String[] args) {
        // Define la cadena de entrada directamente en el
        código
        //input = "num+num"; // Puedes cambiar la cadena para
        probar otros ejemplos
        //input = "+num"; // Puedes cambiar la cadena para
        probar otros ejemplos
        input = "num"; // Puedes cambiar la cadena para
        probar otros ejemplos
        input = input.replaceAll("\\s+", ""); // Eliminar
        espacios en blanco

        if (E() && index == input.length()) {
            System.out.println("Cadena válida según la
            gramática.");
        } else {
            System.out.println("Cadena inválida.");
        }
    }

    // Función para el no terminal E → T E'
    public static boolean E() {
        System.out.println("Procesando E");
        if (T()) { // Verifica si T es válido
            if (Ep()) { // Verifica si E' es válido
                return true;
            }
        }
        return false;
    }

    // Función para el no terminal E' → + T E' | ε
    public static boolean Ep() {
        System.out.println("Procesando E'");
        if (tokenActual() == '+') { // Verifica si hay un
        símbolo "+"
            avanzarToken(); // Avanza el token
            if (T()) { // Verifica si T es válido

```

```
        if (Ep()) { // Recurre para ver si E' es
válido de nuevo
            return true;
        }
    }
    return false;
}
// E' puede ser vacío ( $\epsilon$ ), así que si no hay '+', es
válido
return true;
}

// Función para el no terminal T → num
public static boolean T() {
    System.out.println("Procesando T");
    if (input.startsWith("num", index)) { // Verifica si
"num" está en la posición actual
        avanzarToken(3); // Avanza el índice en 3
posiciones (equivalente a "num")
        return true;
    }
    return false;
}

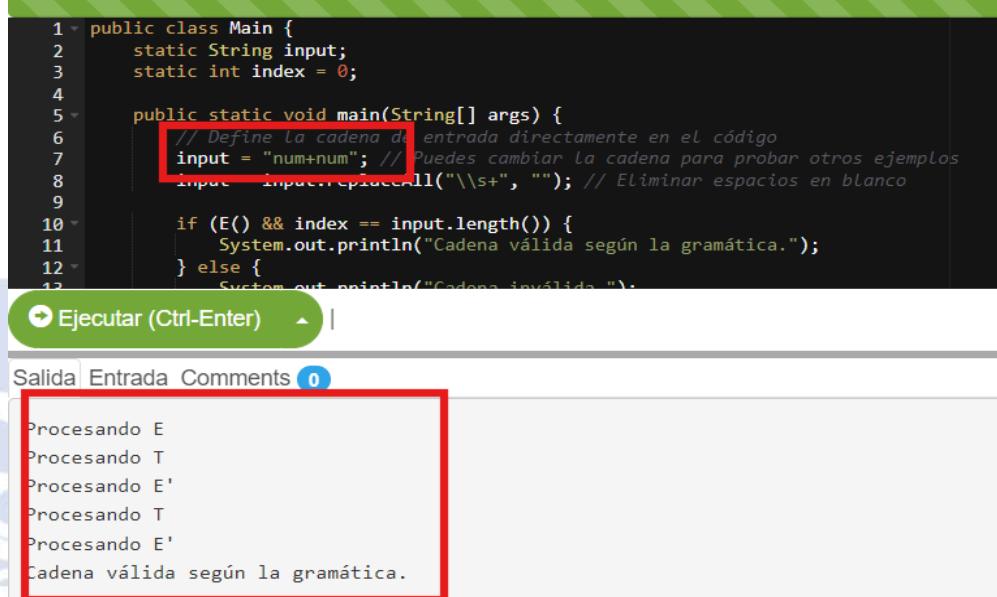
// Avanza al siguiente token de la cadena (o avanza varias
posiciones)
public static void avanzarToken(int cantidad) {
    index += cantidad; // Mueve el índice el número de
posiciones indicadas
}

public static void avanzarToken() {
    avanzarToken(1); // Por defecto avanza 1 token
}

// Devuelve el token actual en la cadena
public static char tokenActual() {
    if (index < input.length()) {
        return input.charAt(index);
    }
    return '\0'; // Retorna carácter nulo si ya se procesó
toda la entrada
}
}
```

## Ejemplos de uso:

1.-Con: "num+num"

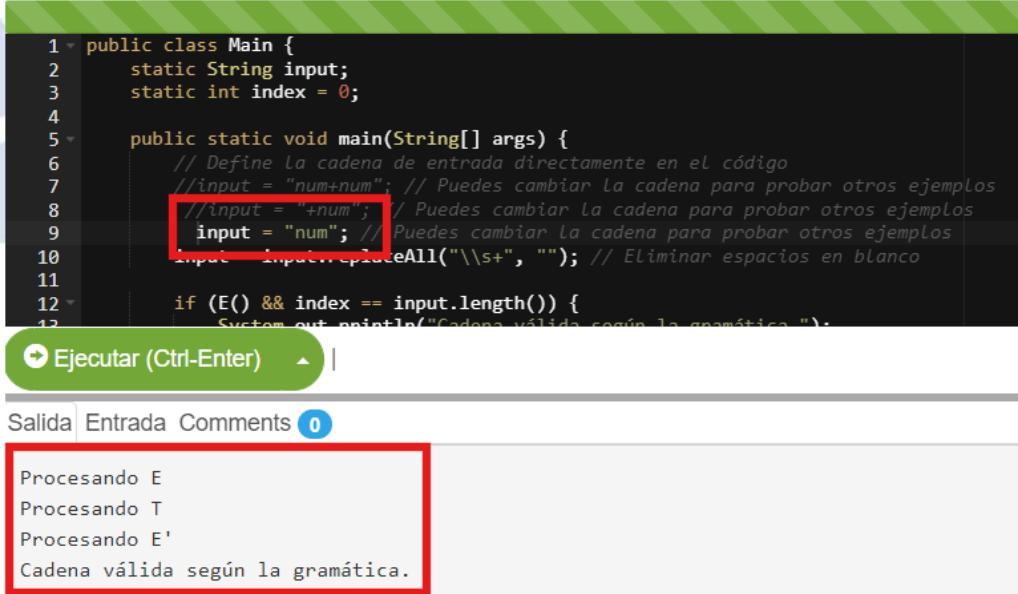


```
1 public class Main {  
2     static String input;  
3     static int index = 0;  
4  
5     public static void main(String[] args) {  
6         // Define la cadena de entrada directamente en el código  
7         input = "num+num"; // Puedes cambiar la cadena para probar otros ejemplos  
8         input = input.replaceAll("\\s+", ""); // Eliminar espacios en blanco  
9  
10    if (E() && index == input.length()) {  
11        System.out.println("Cadena válida según la gramática.");  
12    } else {  
13        System.out.println("Cadena inválida.");  
14    }  
15 }  
16  
17 Ejecutar (Ctrl-Enter) ▾ |
```

Salida Entrada Comments 0

```
Procesando E  
Procesando T  
Procesando E'  
Procesando T  
Procesando E'  
Cadena válida según la gramática.
```

2.- Con: "num"

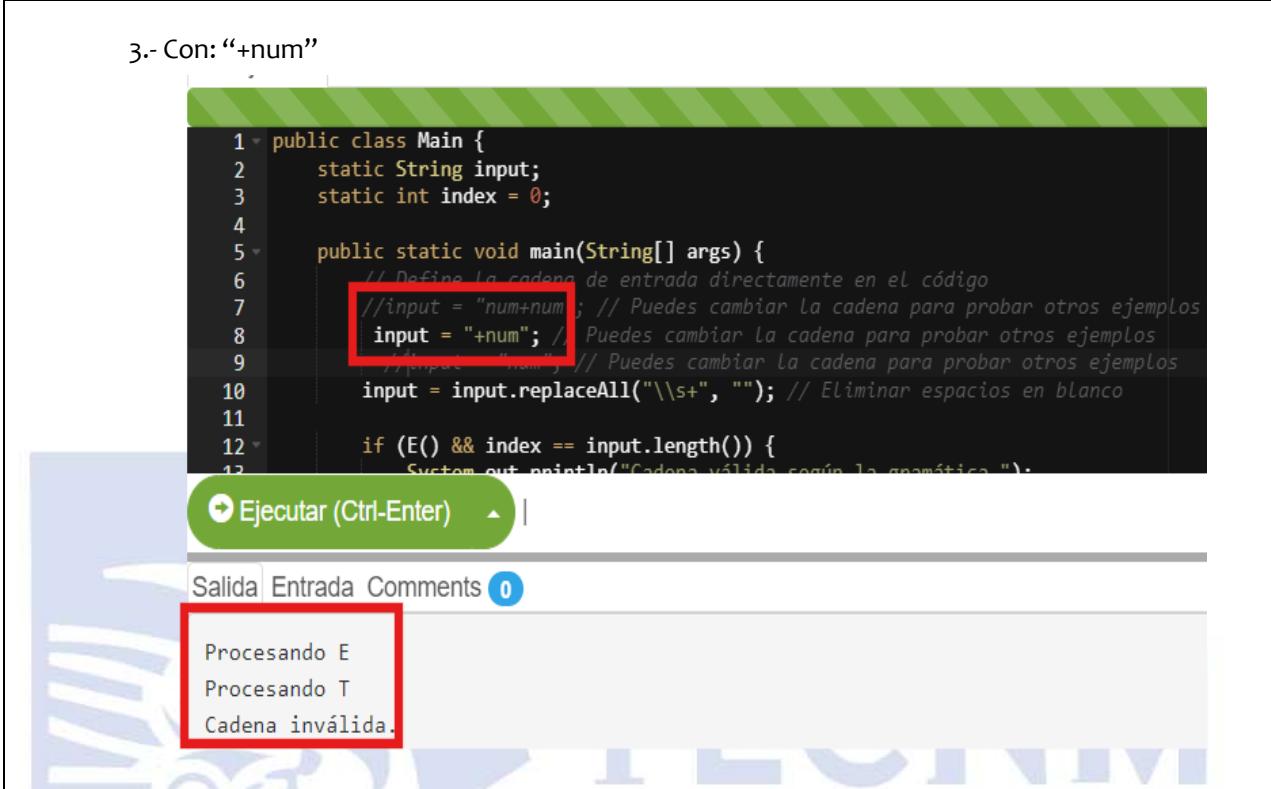


```
1 public class Main {  
2     static String input;  
3     static int index = 0;  
4  
5     public static void main(String[] args) {  
6         // Define la cadena de entrada directamente en el código  
7         //input = "num+num"; // Puedes cambiar la cadena para probar otros ejemplos  
8         //input = "+num"; // Puedes cambiar la cadena para probar otros ejemplos  
9         input = "num"; // Puedes cambiar la cadena para probar otros ejemplos  
10        input = input.replaceAll("\\s+", ""); // Eliminar espacios en blanco  
11  
12    if (E() && index == input.length()) {  
13        System.out.println("Cadena válida según la gramática.");  
14    } else {  
15        System.out.println("Cadena inválida.");  
16    }  
17 }  
18  
19 Ejecutar (Ctrl-Enter) ▾ |
```

Salida Entrada Comments 0

```
Procesando E  
Procesando T  
Procesando E'  
Cadena válida según la gramática.
```

3.- Con: "+num"



```

1 public class Main {
2     static String input;
3     static int index = 0;
4
5     public static void main(String[] args) {
6         // Define la cadena de entrada directamente en el código
7         //input = "num+num"; // Puedes cambiar la cadena para probar otros ejemplos
8         input = "+num"; // Puedes cambiar la cadena para probar otros ejemplos
9         //input = "num"; // Puedes cambiar la cadena para probar otros ejemplos
10        input = input.replaceAll("\\s+", ""); // Eliminar espacios en blanco
11
12        if (E() && index == input.length()) {
13            System.out.println("Cadena válida según la gramática.");
14        } else {
15            System.out.println("Cadena inválida.");
16        }
17    }
18
19    boolean E() {
20        if (index < input.length() && input.charAt(index) == '+') {
21            index++;
22            return T();
23        }
24        return F();
25    }
26
27    boolean T() {
28        if (index < input.length() && input.charAt(index) == 'n') {
29            index++;
30            return F();
31        }
32        return false;
33    }
34
35    boolean F() {
36        if (index < input.length() && input.charAt(index) == 'u') {
37            index++;
38            return true;
39        }
40        return false;
41    }
42 }
  
```

Ejecutar (Ctrl-Enter)

Salida Entrada Comments 0

Procesando E  
Procesando T  
Cadena inválida.

BITÁCORA DE INCIDENCIAS					
PROBLEMA	FECHA	HORA	SOLUCIÓN	FECHA	HORA
El compilador muestra error class is public, should be declared in a file named...	22/09/2024	10:00 AM	Renombrar la clase y el archivo a Main.java.	22/09/2024	10:05 AM
Error de ejecución java.util.NoSuchElementException al usar Scanner.nextLine()	22/09/2024	10:10 AM	Remover el uso de Scanner y definir la cadena de entrada en el código	22/09/2024	10:20 AM
El programa marca "Cadena inválida" para num+num	22/09/2024	10:25 AM	Corregir el reconocimiento del token num y avanzar correctamente el índice	22/09/2024	10:40 AM
El programa no reconoce la cadena completa "num"	22/09/2024	10:45 AM	Modificar la función T() para verificar el token "num" completo y avanzar el índice en 3 posiciones	22/09/2024	10:50 AM

8

### OBSERVACIONES

- **Importancia del manejo de tokens:** Durante el desarrollo, se evidenció la relevancia de identificar correctamente los tokens en la entrada para que el analizador pueda procesarlos de forma adecuada. Al principio, se cometió el error de procesar un solo carácter en lugar de palabras completas como "num", lo que llevó a corregir la función de análisis.
- **Eliminación de espacios en blanco:** En algunas cadenas de entrada, los espacios en blanco pueden generar errores en el análisis. Se implementó la eliminación de espacios para asegurar que la entrada fuera procesada correctamente, independientemente de su formato.
- **Manejo de excepciones:** Se encontró un problema con el uso de Scanner en entornos que no permiten entrada interactiva. Esto resaltó la importancia de adaptar el código al entorno donde se ejecuta, utilizando alternativas como definir la entrada directamente en el código.
- **Limitaciones del analizador:** Aunque el analizador funciona correctamente para expresiones simples como num+num, tiene limitaciones para manejar gramáticas más complejas o entradas mal formadas (como múltiples operadores consecutivos). Este punto sería importante para mejorar en futuras versiones.
- **Necesidad de pruebas exhaustivas:** A lo largo de la práctica, las pruebas con diferentes entradas fueron clave para encontrar errores y ajustar el comportamiento del analizador. Se probó con varias cadenas para asegurar que el código reconociera correctamente las estructuras válidas e inválidas.

9

### ANEXOS

Tuve que tomar referencias sobre como crear la tabla de símbolos.

**En detalle**  
Para  $S' \rightarrow \lambda$

No Terminales	SÍMBOLOS DE ENTRADA					
	i	t	a	e	b	\$
S	$S \rightarrow i S S$		$S \rightarrow a$		$S \rightarrow e S$	
$S'$					$S \rightarrow \lambda$	
C					$C \rightarrow b$	

El conjunto PRIMERO( $\alpha$ ) equivale a PRIMERO( $\lambda$ ) = {  $\lambda$  }, estamos en la regla 2 y hay que ir a SIGUIENTE( $S'$ )= { \$, e }, por tanto M[A, t], será M[ $S'$ , e|\$] =  $S' \rightarrow \lambda$

10

REFERENCIAS BIBLIOGRÁFICAS

- Lajpop, K. [@adiel\_I]. (s/f). Construyendo un analizador sintáctico LL(1). Youtube. Recuperado el 23 de septiembre de 2024, de [https://www.youtube.com/watch?v=ByvrtzG\\_ITw](https://www.youtube.com/watch?v=ByvrtzG_ITw)
- (S/f-a). Cartagena99.com. Recuperado el 23 de septiembre de 2024, de [https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2\\_M4\\_U3\\_T4.pdf](https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2_M4_U3_T4.pdf)
- (S/f-b). Ecured.cu. Recuperado el 23 de septiembre de 2024, de [https://www.ecured.cu/Analizador\\_sintáctico\\_LL](https://www.ecured.cu/Analizador_sintáctico_LL)
- (S/f-c). Uhu.es. Recuperado el 23 de septiembre de 2024, de [https://www.uhu.es/francisco.moreno/gii\\_pl/docs/Tema\\_3.pdf](https://www.uhu.es/francisco.moreno/gii_pl/docs/Tema_3.pdf)



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

DOCENTE DESIGNADO
ISC. RICARDO GONZÁLEZ GONZÁLEZ

PRACTICA No.	NOMBRE DE LA PRACTICA	DURACIÓN ( HORAS )
4	Análisis Sintáctico Descendente	4 horas

1	INTRODUCCIÓN
Un analizador ascendente es un tipo de analizador sintáctico utilizado en la compilación y procesamiento de lenguajes formales, cuya principal función es descomponer una cadena de entrada en sus componentes estructurales según una gramática formal. Estos analizadores construyen el árbol sintáctico de la entrada a partir de las hojas (componentes básicos) hacia la raíz, es decir, trabajan desde la parte más baja del árbol (tokens) y progresan hacia niveles más altos de abstracción, hasta llegar al símbolo inicial de la gramática. El proceso de análisis ascendente es clave en los compiladores, donde se toma una entrada (el código fuente) y se interpreta siguiendo las reglas sintácticas del lenguaje, para después generar código intermedio o traducir a otro formato.	

2	OBJETIVO ( COMPETENCIAS )
Comprender el funcionamiento básico de un analizador ascendente, estudiando su estructura, funcionamiento y las técnicas que emplea para analizar cadenas de entrada según una gramática predefinida.	

3

### MARCO TEÓRICO REFERENCIAL

#### Gramáticas Formales

Una gramática formal es un conjunto de reglas que definen cómo las cadenas de símbolos pueden estructurarse dentro de un lenguaje. Las gramáticas se dividen en:

- **Terminales:** Son los símbolos finales que componen las cadenas del lenguaje.
- **No terminales:** Son símbolos abstractos que pueden ser sustituidos por otros terminales o no terminales.
- **Producciones:** Reglas que definen como un símbolo no terminal puede convertirse en una secuencia de terminales y no terminales.

#### Análisis Sintáctico

El análisis sintáctico es el proceso de determinar si una secuencia de tokens (palabras del lenguaje) se ajusta a las reglas de una gramática formal. Se divide en dos tipos principales:

- **Análisis Descendente:** Comienza desde el símbolo inicial de la gramática y trata de generar la cadena de entrada.
- **Análisis Ascendente:** Parte de los tokens de entrada y trata de construir el árbol sintáctico hacia el símbolo inicial de la gramática.

#### Análisis Ascendente

El análisis ascendente busca reconocer fragmentos de la cadena de entrada que correspondan a las reglas de la gramática, aplicando las producciones de forma inversa, es decir, comenzando por las derivaciones más profundas (tokens) y ascendiendo hasta el símbolo inicial. Este enfoque se basa en dos operaciones fundamentales:

- **Shift:** Cuando el analizador lee un token de la entrada y lo mueve a una pila.
- **Reduce:** Cuando un conjunto de tokens en la pila coincide con una producción de la gramática, se reemplaza por el símbolo no terminal correspondiente.

#### Aplicaciones del Análisis Ascendente

Los analizadores ascendentes se emplean principalmente en la construcción de compiladores y procesadores de lenguajes formales. Entre sus principales aplicaciones están:

- **Generación de árboles sintácticos:** Representan la estructura jerárquica de las cadenas de entrada.
- **Verificación de sintaxis en lenguajes de programación:** Los compiladores utilizan analizadores LR para verificar si el código fuente sigue las reglas gramaticales del lenguaje.
- **Interpretación y compilación de lenguajes:** Una vez generado el árbol sintáctico, este puede ser utilizado para la generación de código intermedio o la ejecución directa de programas.

4

**MATERIALES UTILIZADO**

- Equipo de computo
- Navegador Web (Consulta de información)
- Hojas blancas

5

**REQUISITOS BÁSICOS.**

- Sistema operativo: Windows 2000 / 98 / XP / Vista / 7 / 8 / 10 / 11
- Memoria (RAM): 256 MB
- Espacio disco duro: 100 MB

6

**DESARROLLO DE LA PRÁCTICA ( PASO A PASO )**



## ¿COMO FUNCIONA UN ANALIZADOR ASCENDENTE?

Un analizador ascendente es una técnica utilizada en procesamiento de lenguajes formales y construcción de compiladores, específicamente en el análisis sintáctico, donde el objetivo es determinar si una cadena de entrada pertenece a un lenguaje formal y construir el árbol de derivación correspondiente. A diferencia de un analizador descendente que parte de la regla inicial de la gramática y trata de generar la cadena, el analizador ascendente parte de la cadena de entrada e intenta reducirla a la regla inicial de la gramática.

## FUNCIONAMIENTO DE UN ANALIZADOR ASCENDENTE

En términos generales, el analizador ascendente funciona combinando secuencias de símbolos de la cadena de entrada para formar símbolos no terminales, y luego repite el proceso hasta que la cadena completa se haya reducido a la regla inicial de la gramática.

Pasos básicos del proceso de un analizador ascendente:

1- Lectura de la entrada: El analizador lee la cadena de entrada desde la izquierda hacia la derecha.

2- Reducción: El analizador identifica secuencias de símbolos que coinciden con el lado derecho de alguna producción de la gramática, y las "reduce" al símbolo no terminal correspondiente. Esta reducción es esencialmente el proceso de aplicar las reglas de la gramática en sentido inverso.

3- Reconocimiento de la gramática: Si la cadena puede reducirse exitosamente hasta el símbolo inicial de la gramática, la cadena es aceptada como válida.

4- Manejo de errores: Si el analizador no puede hacer una reducción válida, puede aplicar mecanismos de recuperación de errores para continuar con el análisis o indicar que la cadena no pertenece al lenguaje.

## TIPOS DE ANALIZADORES ASCENDENTES

Existen diferentes implementaciones de analizadores dif. ascendentes:

- LR (Left-to-right, Rightmost derivation): Es uno de los analizadores ascendentes más comunes, que realiza una derivación por la derecha.
- SLR (Simple LR): Una versión más simple del analizador LR que se utiliza cuando la gramática es menos compleja.
- LARL (Look-Ahead LR): Es una optimización de los analizadores LR que combina estados similares, lo que reduce la memoria requerida sin perder potencia de análisis.

### EJEMPLO PRACTICO:

Supongamos una gramática simple que genera expresiones aritméticas

#### 1. Gramática:

- $S \rightarrow E$
- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid id$

#### 2. Entrada:

$id + id * id$

Donde S es la regla inicial,  
E representa una expresión,  
T representa un término,  
F representa un factor y  
id es un identificador  
(una variable o número).

## PROCESO DE ANÁLISIS ASCENDENTE

1- Lectura de entrada: Se empieza con la cadena de entrada:  $id + id * id$ .

2- Reducción:

- Primero, se reduce  $\underline{id}$  a  $\underline{E}$  según la producción  $F \rightarrow id$ .
- Luego, se reduce  $\underline{F}$  a  $\underline{T}$  según la producción  $T \rightarrow F$ .
- Se continua con la reducción de  $\underline{id * id}$  a  $\underline{T * F}$ , que se reduce a  $\underline{T}$ .
- Finalmente, se aplica la producción  $E \rightarrow E + T$ , que permite reducir toda la expresión a  $\underline{E}$ .

3- Aceptación: Si al final del proceso se ha reducido toda la entrada a la regla inicial  $S \rightarrow E$ , entonces la cadena es válida según la gramática.

## VENTAJAS DEL ANÁLISIS ASCENDENTE

1- Mayor cobertura de gramáticas: Los analizadores ascendentes pueden manejar gramáticas más amplias y complejas en comparación con los analizadores descendentes, incluyendo algunas gramáticas que no son adecuadas para los analizadores descendentes predictivos.

2-Detección temprana de errores: El análisis ascendente puede detectar errores a medida que procesa la entrada, lo que facilita la depuración y el manejo de errores.

3-Construcción directa de árboles de derivación: El análisis ascendente permite construir el árbol de derivación a medida que se reduce la entrada, lo que facilita su visualización y optimización.

### DESVENTAJAS DEL ANÁLISIS ASCENDENTE

1-Mayor complejidad en la implementación: A pesar de su capacidad para manejar gramáticas más complejas, los analizadores ascendentes suelen ser más difíciles de implementar que los analizadores descendentes.

2-Uso de memoria: Algunos analizadores ascendentes suelen, como el LR, requerir grandes cantidades de memoria para manejar el conjunto de estados y las tablas de análisis.

### EJEMPLO DE UN ANALIZADOR LR.

El proceso del analizador LR puede ilustrarse mejor mediante un ejemplo de cómo se construye una tabla LR y como se utiliza para una cadena.

Consideramos la gramática:

1- Gramática

$$\begin{array}{l} \bullet S \rightarrow E \\ \bullet E \rightarrow E + T \mid T \\ \bullet T \rightarrow id \end{array}$$

La entrada es: id + id

Tabla LR

Estado	id	+	\$	E	T
0	S5			1	2
1		S4	acc		
2		r2	r2		
4	S5			6	2
5		r3	r3		

Donde S significa "Shift", r significa "reducir" y acc significa "aceptar".

- La cadena de entrada se procesa paso a paso en función de las acciones especificadas en la tabla
- El analizador se desplaza por la entrada (id y +) y realiza reducciones basadas en la producción de la gramática hasta aceptar la cadena.

Este enfoque muestra como un analizador LR realiza análisis sintáctico eficiente mediante una tabla predefinida

5

### BITÁCORA DE INCIDENCIAS

Para el desarrollo de esta practica no se encontraron incidencias

PROBLEMA	FECHA	HORA	SOLUCIÓN	FECHA	HORA

6

### OBSERVACIONES

Importancia del Entendimiento de Gramáticas:

Es fundamental que comprendas la gramática que estás utilizando, ya que el éxito de un analizador ascendente depende de una correcta interpretación de las producciones y la estructura gramatical. Si la gramática no está bien definida o no es adecuada, el analizador no podrá procesar correctamente la entrada.

Importancia de la configuración de la Tabla de Análisis:

Es posible que la mayor dificultad radique en la construcción o comprensión de la tabla de análisis LR. Asegúrate de comprender cómo se generan los estados y cómo se determinan las acciones (shift, reduce, aceptar, o error) para cada combinación de token y estado. Un error en esta tabla puede llevar a bucles infinitos o fallos en el análisis.

7

### ANEXOS

Una de las fuentes de información más útiles usadas para esta practica fue el siguiente video:

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

Con ejemplos vistos en este video se pudo entender mejor el como funciona el análisis sintáctico ascendente.

**Ejemplo:**

- ✓  $E \rightarrow E + T$  ?
- ✗ |  $T$
- ✗  $T \rightarrow T * F$
- ✗ |  $F$
- ✗  $F \rightarrow (E)$
- ✗ |  $num$

Analizar:  $5+2*5$

Pila	Entrada	Acción
\$	5+2*5	Shift
\$5	+2*5	Shift S ->
\$5*	2*5	Reduce S->
\$5*	2*	Shift S ->
\$5*	2	Shift S ->
\$5*		Shift S ->
\$5*		Accept

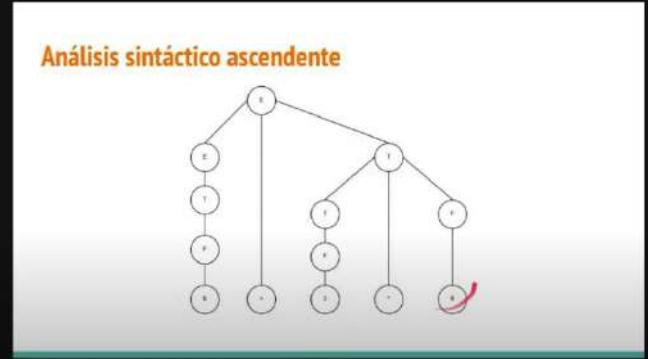
$\Leftarrow$

$\Rightarrow E \rightarrow E$   
 $| E \rightarrow E$   
 $| E \rightarrow num$



KEVIN ADIEL LA JOPOP AJPACA JA

100%  
100%  
100%



Análisis sintáctico ascendente



KEVIN LAJPOP AJACAJA

7

#### REFERENCIAS BIBLIOGRÁFICAS

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, techniques, and tools (2nd ed.). Pearson Education.
- Fischer, C. N., & LeBlanc, R. J. (1991). Crafting a compiler. Benjamin/Cummings Publishing.
- Appel, A. W., & Palsberg, J. (2002). Modern compiler implementation in C. Cambridge University Press.
- Kevin Lajpop. (2021, October 20). Introducción al análisis sintáctico ascendente [Video]. YouTube. <https://www.youtube.com/watch?v=wINHIfxswOs>



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

DOCENTE DESIGNADO
ISC. RICARDO GONZÁLEZ GONZÁLEZ

PRACTICA No.	NOMBRE DE LA PRACTICA	DURACIÓN ( HORAS )
5	¿ CUÁLES SON LOS OBJETIVOS Y LAS FUNCIONES DE UN ANALIZADOR SINTÁCTICO ?	3 (HORAS)

1	INTRODUCCIÓN
Un analizador sintáctico es una herramienta clave encargada de verificar que el código fuente cumpla con las reglas gramaticales del lenguaje de programación utilizado. Su función principal es procesar la secuencia de tokens generada por el analizador léxico y determinar si la estructura es válida. Este proceso permite identificar errores y generar una estructura de datos que el compilador utilizará para producir el código ejecutable.	

2	OBJETIVO ( COMPETENCIAS )
<ul style="list-style-type: none"><li>• <b>Comprender</b> el funcionamiento y la importancia de un analizador sintáctico en el proceso de compilación.</li><li>• <b>Identificar</b> los diferentes tipos de analizadores sintácticos y sus aplicaciones.</li><li>• <b>Aplicar</b> conocimientos teóricos para implementar un analizador sintáctico básico.</li><li>• <b>Evaluar</b> la eficiencia y precisión de diferentes métodos de análisis sintáctico</li></ul>	

3

### **MARCO TEÓRICO REFERENCIAL**

- El analizador sintáctico es parte esencial de la segunda fase de un compilador. Recibe una cadena de tokens, la cual es una representación del código fuente descompuesto en sus componentes más básicos, y organiza esos tokens de acuerdo con las reglas gramaticales del lenguaje.
- **1. Tipos de Analizadores Sintácticos**  
**Analizador Ascendente:** Construyen el árbol sintáctico desde las hojas hasta la raíz. Un ejemplo es el algoritmo de análisis LR.  
**Analizador Descendente:** Comienzan desde la raíz del árbol sintáctico e intentan derivar la entrada desde la parte superior, siguiendo la gramática del lenguaje. El algoritmo LL es un ejemplo.
- **2. Objetivos de un Analizador Sintáctico**  
Verificar si la secuencia de tokens cumple con las reglas gramaticales.  
Construir una representación estructurada (como un árbol sintáctico) del código fuente.  
Proporcionar retroalimentación de errores de sintaxis al programador.
- **3. Funciones del Analizador Sintáctico**  
**Detección de errores:** El analizador debe ser capaz de detectar cualquier error sintáctico y proporcionar mensajes útiles.  
**Construcción del árbol sintáctico:** Generar una representación jerárquica de la estructura del código, lo cual es vital para el siguiente paso en la compilación: la generación de código intermedio.  
**Compatibilidad con múltiples lenguajes:** Los analizadores sintácticos se diseñan para adaptarse a las características gramaticales de lenguajes específicos.
- **4. Gramáticas Libres de Contexto**  
El análisis sintáctico se basa generalmente en gramáticas libres de contexto (CFG), que son las reglas formales que describen cómo pueden combinarse los elementos del lenguaje. Cada regla define como un símbolo no terminal puede derivarse en una secuencia de terminales y no terminales.

4

**MATERIALES UTILIZADO**

- **Lenguaje de Programación:**
  - **Java:** Utilizado para implementar el código del analizador sintáctico.
- **Entorno de Desarrollo:**
  - **IDE:** Eclipse, IntelliJ IDEA o NetBeans para escribir y depurar el código.

5

**REQUISITOS BÁSICOS.**

- Conocimientos de gramáticas formales y la teoría de lenguajes y autómatas.
- Conocimiento básico de Java
- Conocimientos básicos de analizadores léxicos
- Conocimientos básicos de analizadores sintáctico



6

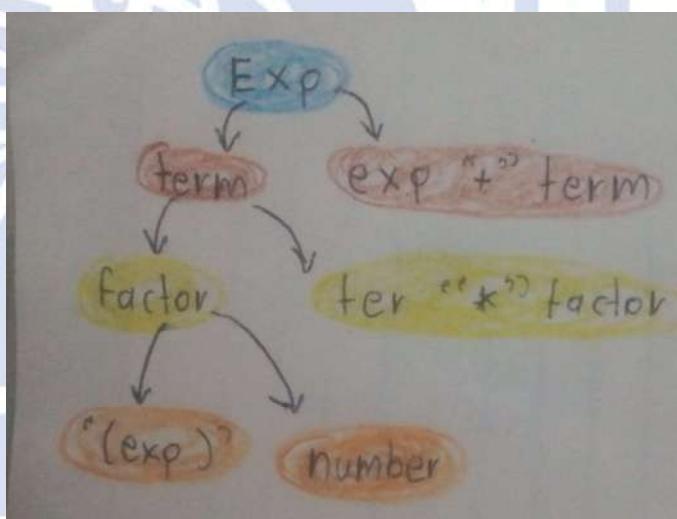
DESARROLLO DE LA PRÁCTICA ( PASO A PASO )

1. Definición del Lenguaje

Selección de lenguaje simple y definición de su gramática formal utilizando una notación como BNF (Backus-Naur Form). Por ejemplo, calculadora básica que maneje sumas y multiplicaciones.

```
<exp> ::= <term> | <exp> "+" <term>  
<term> ::= <factor> | <term> "*" <factor>  
<factor> ::= "(" <exp> ")" | number
```

Diagrama de Sintaxis



## 2. Implementación del Analizador

Crea un analizador léxico que divide el código fuente en tokens. Los tokens son las unidades básicas del lenguaje, como números, operadores y paréntesis.

### Construcción de Árbol de derivación

algoritmo que utiliza la gramática definida para construir un árbol de derivación a partir de los tokens.

INICIO

  FUNCION AnalizarSintaxis(expresion)

    PILA operadores

    PILA operandos

    PARA cada carácter en expresion HACER

      SI carácter ES un número ENTONCES

        APILAR(operandos, carácter)

      SINO SI carácter ES un operador (+,\*) ENTONCES

        APILAR(operadores, carácter)

      SINO

        IMPRIMIR "Error: Carácter inválido"

        RETORNAR FALSO

      FIN SI

    FIN PARA

    MIENTRAS operadores NO ESTÁ VACÍA HACER

      operador = DESAPILAR(operadores)

      operando2 = DESAPILAR(operandos)

      operando1 = DESAPILAR(operandos)

      resultado = Evaluar(operador, operando1, operando2)

      APILAR(operandos, resultado)

    FIN MIENTRAS

    SI operandos TIENE UN SOLO ELEMENTO ENTONCES

      IMPRIMIR "Expresión válida"

      RETORNAR VERDADERO

    SINO

      IMPRIMIR "Error: Expresión inválida"

      RETORNAR FALSO

    FIN SI

  FIN FUNCION

  FUNCION Evaluar(operador, operando1, operando2)

    SI operador ES '+' ENTONCES

      RETORNAR operando1 + operando2

    SINO SI operador ES '-' ENTONCES

      RETORNAR operando1 - operando2

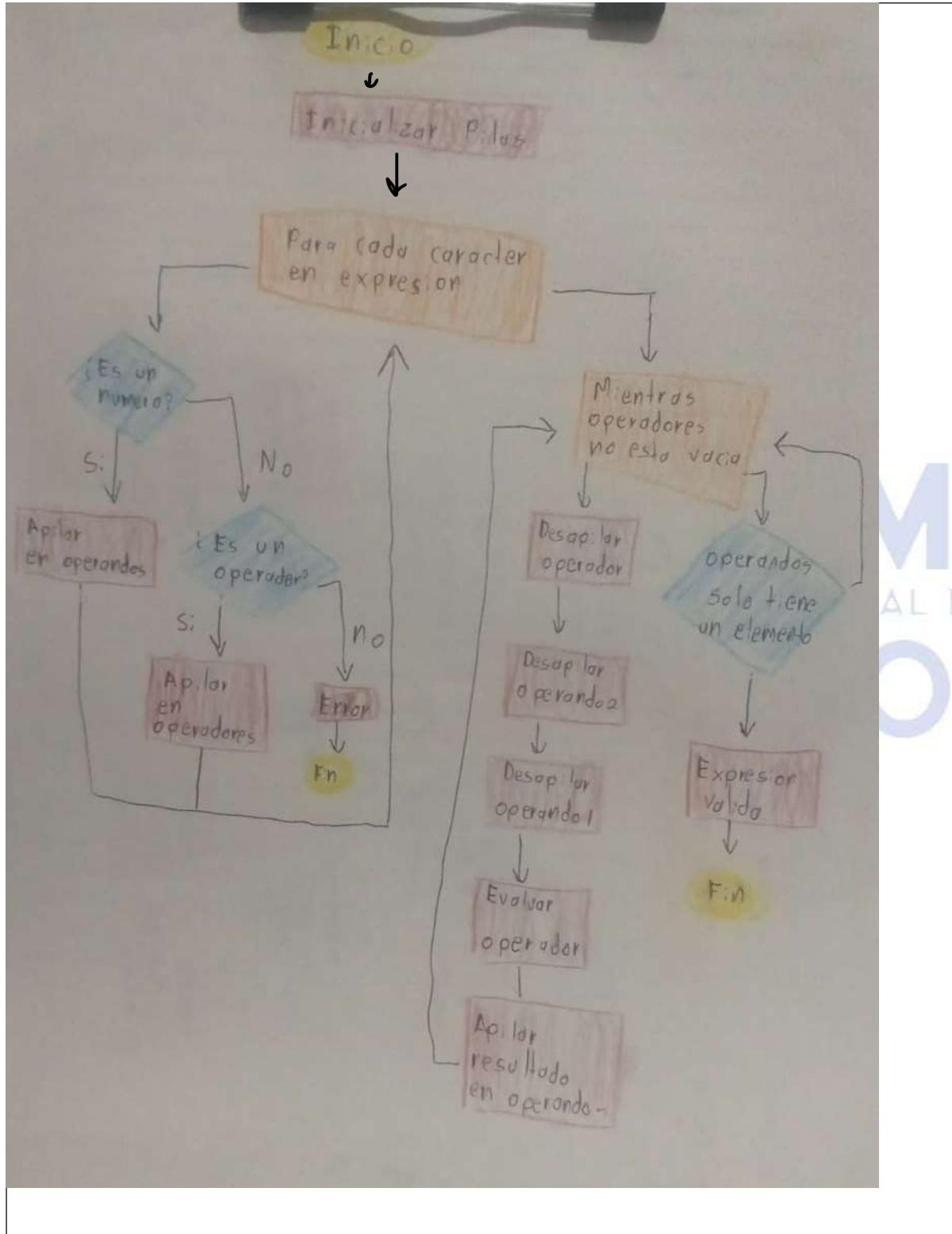
    SINO SI operador ES '\*' ENTONCES

      RETORNAR operando1 \* operando2

    FIN SI

  FIN FUNCION

FIN



### 3. Pruebas y Validación

- a. Para esta prueba se realizara en java con la prueba (3+5)

```
1  import java.util.Stack;
2
3  public class Main {
4      public static void main(String[] args) {
5          String expresion = "3+5";
6          if (analizarSintaxis(expresion)) {
7              System.out.println("Expresión válida");
8          } else {
9              System.out.println("Error: Expresión inválida");
10         }
11     }
12
13     public static boolean analizarSintaxis(String expresion) {
14         Stack<Character> operadores = new Stack<>();
15         Stack<Integer> operandos = new Stack<>();
16
17         for (char caracter : expresion.toCharArray()) {
18             if (Character.isDigit(caracter)) {
19                 operandos.push(Character.getNumericValue(caracter));
20             } else if (esOperador(caracter)) {
21                 operadores.push(caracter);
22             } else {
23                 System.out.println("Error: Carácter inválido");
24                 return false;
25             }
26         }
27
28         while (!operadores.isEmpty()) {
29             char operador = operadores.pop();
30             int operando2 = operandos.pop();
31             int operando1 = operandos.pop();
32             int resultado = evaluar(operador, operando1, operando2);
33             operandos.push(resultado);
34         }
35
36         return operandos.size() == 1;
37     }
38
39     public static boolean esOperador(char caracter) {
40         return caracter == '+' || caracter == '*';
41     }
42
43     public static int evaluar(char operador, int operando1, int operando2) {
44         switch (operador) {
45             case '+':
46                 return operando1 + operando2;
47             case '*':
48                 return operando1 * operando2;
49             default:
```

```
46         return operando1 - operando2;
47     default:
48         throw new IllegalArgumentException("Operador inválido: " + operador);
49     }
50 }
51 }
52 }
53 }
54 }
```

Salida Entrada Comments 0

Expresión válida



6 | OBSERVACIONES

Algunas de las cosas que se pudieron implementar fueron:

- **Definir mejor las fases de implementación:** En el desarrollo de la práctica, sería recomendable separar más claramente las etapas. Es decir, la construcción de la tabla de parsing y su implementación podrían desarrollarse con más detalle para que quede completamente claro cómo cada una de las partes contribuye al proceso de análisis sintáctico.
- **Retroalimentación en caso de errores:** Sería beneficioso incluir cómo el analizador reporta errores y las estrategias comunes de recuperación de errores sintácticos. Este aspecto es fundamental en los analizadores sintácticos reales, ya que no solo verifican el código, sino que también intentan continuar con el análisis para proporcionar un informe detallado.

7 ANEXOS

Para la elaboración de esta práctica recurrió al siguiente material: es una página de la universidad Carlos III de Madrid.

---

UNIVERSIDAD CARLOS III DE MADRID - DEPARTAMENTO DE INGENIERÍA TELEMÁTICA

---

[Localización](#) | [Personal](#) | [Docencia](#)

---

Bibliografía

- Carlos III University of Madrid. (s/f). Práctica 1: Análisis lóxico y sintáctico. Uc3m.es. Recuperado el 21 de septiembre de 2024, de <https://www.it.uc3m.es/luis/pfat/p1/>
- Choréño Portillo Alain Alejandro Navarrete Moreno Rodrigo. (2022, abril 24). Analizador sintáctico. Coursesidekick.com. <https://www.coursesidekick.com/computer-science/6132610>
- Todo lo que necesitas saber sobre los analizadores sintácticos: Guía completa y paso a paso. (2023, noviembre 20). Aprendomax.com. <https://aprendomax.com/recursos/analizadores-sintacticos/>



**TECNM**  
TECNOLOGICO NACIONAL DE  
**MÉXICO**

CARRERA	NOMBRE DE LA ASIGNATURA
INGENIERIA EN SISTEMAS COMPUTACIONALES	LENGUAJES Y AUTOMATAS 2

DOCENTE DESIGNADO
ISC. RICARDO GONZÁLEZ GONZÁLEZ

PRACTICA No.	NOMBRE DE LA PRACTICA	DURACIÓN ( HORAS )
6	¿CÓMO FUNCIONA EL GENERADOR DE CÓDIGO PARA ANALIZADORES SINTÁCTICOS YACC?	3 (HORAS)

1

### INTRODUCCIÓN

El análisis sintáctico es una de las etapas fundamentales en la construcción de compiladores e intérpretes de lenguajes de programación. En esta fase, se analiza la estructura de un programa de acuerdo con las reglas de la gramática del lenguaje. Yacc (Yet Another Compiler Compiler) es una herramienta que permite generar automáticamente analizadores sintácticos basados en una gramática formal definida por el usuario.

Yacc es comúnmente utilizado en conjunto con Lex, una herramienta para el análisis léxico. Mientras Lex divide el texto de entrada en tokens, Yacc utiliza estos tokens para construir una estructura jerárquica, siguiendo las reglas de la gramática definida.

El propósito de esta práctica es comprender cómo funciona Yacc, cómo se define una gramática, y cómo se genera el código necesario para crear un analizador sintáctico. A través de ejemplos prácticos y la implementación de pequeñas gramáticas, se explorarán las bases del análisis sintáctico y su rol en el desarrollo de compiladores.

2

### OBJETIVO ( COMPETENCIAS )

- Comprender el funcionamiento del generador de código Yacc.
- Desarrollar habilidades para construir analizadores sintácticos.
- Implementar gramáticas para el análisis de lenguajes.
- Desarrollar competencias en el uso de herramientas de desarrollo como Yacc y Lex.

3

### MARCO TEÓRICO REFERENCIAL

✓ **¿Qué es Yacc?**

Yacc (Yet Another Compiler Compiler) es una herramienta que genera un analizador sintáctico a partir de una gramática definida. Es usada junto con Lex, que se encarga del análisis léxico.

✓ **¿Cómo funciona?**

Yacc toma una gramática formal y genera un parser que sigue la técnica LR (Left-to-right, Rightmost derivation). El parser generado analiza el código fuente de un lenguaje y crea una representación estructurada que puede ser usada para la compilación o interpretación.

✓ **Relación con Lex:**

Yacc trabaja junto con Lex, que se encarga de dividir la entrada en tokens. Los tokens son luego analizados por el parser generado por Yacc para formar la estructura sintáctica.

✓ **Estructura de un archivo Yacc:**

- ⊕ Declaraciones: Define los tokens y tipos de datos.
- ⊕ Reglas: Gramática del lenguaje.
- ⊕ Código adicional: Funciones en C/C++ que el parser usa.

4

### MATERIALES UTILIZADO

• **Software:**

- † Yacc o Bison (versión moderna de Yacc).
- † Lex o Flex (para el análisis léxico).
- † Un compilador C/C++ (como GCC).
- † Un editor de texto (como VSCode o Vim).

• **Hardware:**

- † Tu computadora con WSL, si estás trabajando en Linux, o en un entorno Linux directamente.

• **Documentación:**

- † Manuales o guías de Yacc/Bison.
- † Gramáticas para el lenguaje que estés analizando.

5

### REQUISITOS BÁSICOS.

- Conocimientos de gramáticas formales y la teoría de lenguajes y autómatas.
- Familiaridad con análisis léxico y el uso de herramientas como Lex o Flex.
- Conocimiento de lenguajes de programación C o C++ para entender el código generado por Yacc.
- Configuración de un entorno de desarrollo con las herramientas necesarias (WSL, compilador GCC, Yacc/Bison).

6

### DESARROLLO DE LA PRÁCTICA (PASO A PASO)

- **Archivo calc.y (Gramática con YACC)**
- Este archivo define la gramática que el analizador sintáctico usará para reconocer y evaluar expresiones aritméticas. Utiliza las directivas de YACC/Bison para generar el código C que ejecutará la lógica de parsing.

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
  
int yylex(void); // Declaración de la función de análisis  
léxico  
void yyerror(const char *s); // Función de manejo de  
errores  
%}  
  
%token NUM // Definición del token para números  
  
%left '+' '-' // Establece que '+' y '-' son operadores con  
la misma precedencia (izquierda)  
%left '*' '/' // Establece que '*' y '/' tienen mayor  
precedencia que '+' y '-'  
  
%% // Inicio de la parte de reglas de la gramática  
  
input:  
    | input line // Regla que dice que la entrada puede ser  
    vacía o puede contener líneas de expresiones  
    ;  
  
line:  
    expr '\n' { printf("Resultado: %d\n", $1); } // Evalúa  
    la expresión y muestra el resultado  
    ;  
  
expr:  
    expr '+' expr { $$ = $1 + $3; } // Suma: $$ es el valor  
    de la expresión completa, $1 y $3 son los operandos  
    | expr '-' expr { $$ = $1 - $3; } // Resta  
    | expr '*' expr { $$ = $1 * $3; } // Multiplicación
```

```
| expr '/' expr { $$ = $1 / $3; } // División
| NUM { $$ = $1; } // El número en sí
;

%% // Fin de la gramática

int main(void) {
    printf("Ingrese expresiones (Ctrl+D para salir):\n");
    return yyparse(); // Llama a la función de análisis
sintáctico generada
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s); // Muestra los
errores en caso de que ocurra uno
}
```

**Explicación:**

Bloque %{ %}: Dentro de este bloque, se incluyen las declaraciones de C necesarias para el programa, como las funciones yylex() y yyerror().

**Tokens:**

NUM es el token que representa un número entero. Este token será devuelto por el analizador léxico cuando encuentre números.

**Precedencia y Asociatividad:**

%left '+' '-': Define que los operadores + y - tienen asociatividad izquierda, lo que significa que en una expresión como  $3 - 2 + 1$ , se evaluará como  $((3 - 2) + 1)$ .

%left '\*' '/': Define que la multiplicación y la división tienen mayor precedencia que la suma y la resta, lo que sigue las reglas habituales de las operaciones matemáticas.

**Reglas de Gramática:**

input: Puede ser una secuencia de líneas o vacío.

line: Representa una línea que contiene una expresión aritmética seguida por un salto de línea ('\n'). Cuando se encuentra una línea, se evalúa y muestra el resultado.

expr: Define las reglas para las expresiones aritméticas, indicando cómo se deben combinar los operandos y operadores. Por ejemplo:

expr '+' expr: Esta regla especifica cómo sumar dos expresiones.

NUM: El valor del número se asigna directamente a la variable que representa el resultado de la expresión.

**Funciones:**

int main(void): El programa empieza aquí, mostrando un mensaje e invocando yyparse(), que es la función generada por YACC que inicia el análisis sintáctico.

void yyerror(const char \*s): Esta función maneja los errores de parsing, mostrando un mensaje de error en caso de que se produzcan.

- **Archivo calc.l (Analizador Léxico con Lex)**

Este archivo se encarga de analizar el texto de entrada y dividirlo en "tokens" que serán utilizados por el analizador sintáctico. En este caso, el analizador léxico reconoce números, espacios y operadores.

```
%{  
#include "calc.tab.h" // Incluye el archivo de encabezado  
generado por Bison  
%}  
  
%% // Inicio de las reglas léxicas  
  
[0-9]+ { yyval = atoi(yytext); return NUM; } //  
Reconoce números y los convierte a enteros  
[ \t] ; // Ignora espacios y tabulaciones  
\n { return '\n'; } // Retorna el token de salto de  
línea  
. { return yytext[0]; } // Retorna cualquier otro  
carácter (operadores, etc.)  
  
%% // Fin de las reglas léxicas  
  
int yywrap(void) {  
    return 1; // Indica que no hay más entradas  
}
```

**Explicación:**

Bloque %{ %}: Incluye el archivo calc.tab.h, que contiene las definiciones de tokens generadas por Bison. Esta inclusión es necesaria para que el código Lex reconozca los tokens como NUM.

**Reglas Léxicas:**

[0-9]+: Esta expresión regular reconoce secuencias de uno o más dígitos. Cuando encuentra un número, lo convierte a un entero utilizando atoi(yytext) y lo asocia al token NUM.

[ \t]: Ignora los espacios y las tabulaciones, lo que es útil para que el analizador léxico no los considere tokens relevantes.

\n: Detecta un salto de línea y retorna el token correspondiente. Esto es importante para permitir que el usuario ingrese varias expresiones, una por línea.

.: Cualquier otro carácter que no sea un número, espacio o salto de línea se retorna tal cual, lo que permite procesar operadores como +, -, \*, y /.

**Función yywrap():**

Esta función indica que no hay más entradas que analizar cuando el analizador llega al final del archivo de entrada. Retorna 1 para finalizar el proceso de análisis léxico.

**Ejecución del programa:**

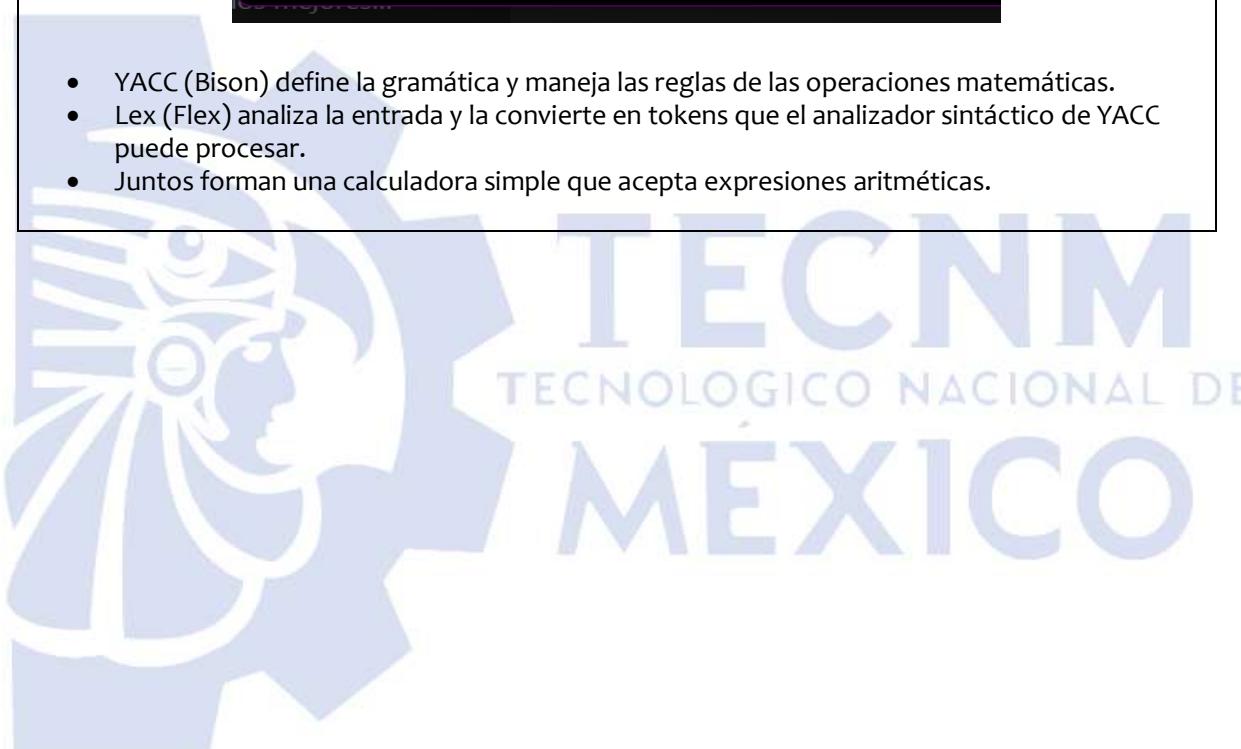
```
dust@LAPTOP-GG069FL4:~/automatas$ bison -d calc.y  
dust@LAPTOP-GG069FL4:~/automatas$ flex calc.l  
dust@LAPTOP-GG069FL4:~/automatas$ gcc -o calc calc.tab.c lex.yy.c -lfl
```

- Bison genera los archivos calc.tab.c y calc.tab.h a partir de calc.y.
- Flex genera el archivo lex.yy.c a partir de calc.l.
- Finalmente, se compilan ambos (calc.tab.c y lex.yy.c) usando GCC para crear el ejecutable calc.

**Ejemplo de uso:**

```
dust@LAPTOP-GG069FL4:~/automatas$ ./calc
Ingrrese expresiones (Ctrl+D para salir):
3+5
Resultado: 8
35+8
Resultado: 43
l
Error: syntax error
dust@LAPTOP-GG069FL4:~/automatas$ |
```

- YACC (Bison) define la gramática y maneja las reglas de las operaciones matemáticas.
- Lex (Flex) analiza la entrada y la convierte en tokens que el analizador sintáctico de YACC puede procesar.
- Juntos forman una calculadora simple que acepta expresiones aritméticas.



5

## BITÁCORA DE INCIDENCIAS

PROBLEMA	FECHA	HORA	SOLUCIÓN	FECHA	HORA
y.tab.h no encontrado al compilar.	22/0 9/20 24	10:15 AM	Se corrigió el nombre del archivo de inclusión.	22/0 9/20 24	10:45 AM
Conflicto shift/reduce al generar el parser.	22/0 9/20 24	11:00 AM	Se añadieron directivas de precedencia en YACC.	22/0 9/20 24	11:30 AM
Error al reconocer operadores en calc.l.	22/0 9/20 24	12:00 PM	Se ajustaron las reglas para procesar operadores.	22/0 9/20 24	12:30 PM
Falta de respuesta al procesar números grandes.	22/0 9/20 24	1:00 PM	Se mejoró el manejo de números en calc.l.	22/0 9/20 24	1:30 PM
Problema con el Ctrl+D para finalizar entradas.	22/0 9/20 24	2:00 PM	Se ajustó el control de final de entrada.	22/0 9/20 24	2:20 P

6

## OBSERVACIONES

- Conflictos de Desplazamiento/Reducción: Durante el desarrollo de la práctica, se identificaron varios conflictos de desplazamiento/reducción en la gramática YACC. Esto resaltó la importancia de definir correctamente la precedencia y la asociatividad de los operadores, ya que una gramática ambigua puede generar múltiples interpretaciones para las mismas expresiones.
- Importancia de las Directivas en YACC: El uso de las directivas %left y %right fue crucial para resolver los conflictos de precedencia entre operadores aritméticos. Esto demostró que el manejo adecuado de la precedencia es clave para el análisis sintáctico correcto de expresiones matemáticas.
- Inclusión Correcta de Archivos: Al trabajar con Flex y YACC, es fundamental asegurarse de que los archivos generados por ambos (como calc.tab.h) estén correctamente incluidos en el archivo de Lex (calc.l). Cualquier error en la inclusión de estos archivos resultó en fallos durante la compilación.
- Optimización de la Gramática: El proceso de depuración de la gramática permitió una mejor comprensión de cómo simplificar las reglas para evitar ambigüedades y generar un parser más eficiente.
- Flexibilidad del Analizador Léxico: Flex mostró ser bastante flexible al definir patrones simples para reconocer números y operadores. Sin embargo, se identificó la importancia de manejar de manera explícita los espacios en blanco y los saltos de línea para evitar problemas en el reconocimiento de las entradas.
- Manejo de Errores: La función yyerror() proporcionada por YACC permitió manejar los errores en el análisis sintáctico de manera eficiente, aunque es importante mejorar los mensajes de error para que sean más descriptivos y útiles para el usuario final.

- Desempeño General: La práctica demostró que con una buena integración entre Flex y YACC, es posible construir herramientas de análisis de expresiones aritméticas eficientes y personalizadas. Sin embargo, el manejo de expresiones complejas y números grandes podría mejorarse para aumentar la robustez del programa.

7

## ANEXOS

Para la elaboración de esta práctica recurrió a los siguientes materiales: ya que fueron de mucha ayuda porque opte por usar la distribución de Linux para Windows y esta página web tiene ejemplos para crear el programa usando yacc :

*Pre-requisitos*

Para este ejemplo necesitamos las siguientes herramientas:

- Compilador GCC
- Visual Studio Code (o cualquier editor de texto de nuestro agrado)

*Instalación y configuración de las herramientas*

Lo primero que haremos será instalar Lex, para ello abrimos una terminal, en Ubuntu puede hacerse con la combinación de teclas Ctrl + Alt + t o en Aplicaciones → Accesorios → Terminal, una vez abierta la terminal ingresamos el comando:

```
1. sudo apt-get install flex
```

Autenticamos ingresando nuestra contraseña y aceptamos la descarga e instalación, con esto quedará instalado Lex. Como nos pudimos dar cuenta, la instalación de Lex se hace a través Flex que es otra herramienta de análisis léxico.

7

## REFERENCIAS BIBLIOGRÁFICAS

- AIX 7.3. (2023, marzo 24). Ibm.com. <https://www.ibm.com/docs/es/aix/7.3?topic=concepts-lex-yacc-program-information>
- de autómatas y lenguajes formales Alma María Pisabarro Marrón, T. (s/f). El generador de analizadores sintácticos Yacc IV. Uva.es. Recuperado el 23 de septiembre de 2024, de <https://www.infor.uva.es/~mluisa/talf/docs/lab0/L11.pdf>
- Mi primer proyecto utilizando Yacc y Lex. (s/f). Erick Navarro. Recuperado el 23 de septiembre de 2024, de <https://ericknavarro.io/2020/10/01/27-Mi-primer-proyecto-utilizando-Yacc-y-Lex/>
- Universidad de Guanajuato. (2022, julio 16). Clase digital 4. Herramientas de software: Yacc/Bison. Recursos Educativos Abiertos; Sistema Universitario de Multimodalidad Educativo (SUME) - Universidad de Guanajuato. <https://blogs.ugto.mx/rea/clase-digital-4-herramientas-de-software-yacc-bison/>

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

DOCENTE DESIGNADO
ISC. RICARDO GONZÁLEZ GONZÁLEZ

PRACTICA No.	NOMBRE DE LA PRACTICA	DURACIÓN ( HORAS )
5	Análisis Sintáctico – Bison y Características de analizadores sintácticos.	6 horas

1

### **INTRODUCCIÓN**

Yacc y Bison son herramientas usadas en la construcción de compiladores y analizadores sintácticos los cuales procesan lenguajes de programación y otros lenguajes formales. Las dos son herramientas que ayudan en la automatización de la generación de código que se usa para analizar la estructura gramatical de un lenguaje. Son herramientas muy importantes y fundamentales en el desarrollo de analizadores sintácticos y compiladores.

2

### **OBJETIVO ( COMPETENCIAS )**

Crear un programa que demuestre de forma práctica el funcionamiento de Bison en la creación de analizadores sintácticos, así como describir las características de Yacc y Bison.

3

### **MARCO TEÓRICO REFERENCIAL**

Bison es un generador de analizadores sintácticos a partir de una especificación de gramática escrita en un formato muy parecido a la Notación de Backus-Naur (BNF). Como generador de analizadores sintácticos tiene importancia ya que estos son un componente crucial en compiladores e intérpretes de lenguajes, esto porque procesan la estructura sintáctica del código fuente o cualquier entrada secuencial que siga una gramática definida.

Bison es parte de la familia de herramientas GNU, esta diseñado para tener compatibilidad con Yacc el cual es otro generador de analizadores sintácticos. Bison toma una gramática formal, descrita en un archivo fuente y genera un código en C, C++ o Java ya que puede analizar secuencias de símbolos (tokens) que comprueban si se siguen las reglas de la gramática.

Bison trabaja con gramáticas libres de contexto, por previas investigaciones ya se conoce los elementos de las mismas los cuales son:

- Símbolos terminales: tokens del lenguaje, representan elementos básicos como números, operadores, palabras clave, etc.
- Símbolos no terminales: Abstracciones de estructuras sintácticas más complejas, como expresiones.
- Reglas de producción: Reglas que definen como se combinan símbolos terminales y no terminales para formar expresiones válidas en el lenguaje.

Algunos componentes clave en Bison son los tokens, los cuales representan elementos básicos en el análisis sintáctico como ya se mencionó anteriormente. Los tokens se definen en conjunto a herramientas como Flex o Lex en las definiciones del archivo .y en Bison. Al identificar un token, Bison asocia una acción semántica a cada regla de producción. Dichas acciones son fragmentos de código en C que permitirán que el programa haga algo útil con las expresiones que se procesan, como evaluarlas o transformarlas.

Otro componente clave es la resolución de conflictos usando un mecanismo basado en precedencias y asociaciones definidas por el programador. Las precedencias como se vio en la teoría determinan el operador con mayor jerarquía, la asociatividad define si un operador se evalúa de izquierda a derecha o viceversa.

El flujo de trabajo que presenta en conjunto a Flex es el siguiente:

- Flex genera el archivo de código C para el analizador léxico.
- Bison genera el archivo de código C para el analizador sintáctico
- Los archivos son compilados juntos para la creación del programa final.

4

### **MATERIALES UTILIZADO**

- Equipo de computo
- Navegador Web (para consulta de información)
- Bison
- Flex
- Visual Studio

5

**REQUISITOS BÁSICOS.**

- Sistema operativo: Windows 2000 / 98 / XP / Vista / 7 / 8 / 10 / 11
- Memoria (RAM): 256 MB
- Espacio disco duro: 100 MB

6

**DESARROLLO DE LA PRÁCTICA ( PASO A PASO )**

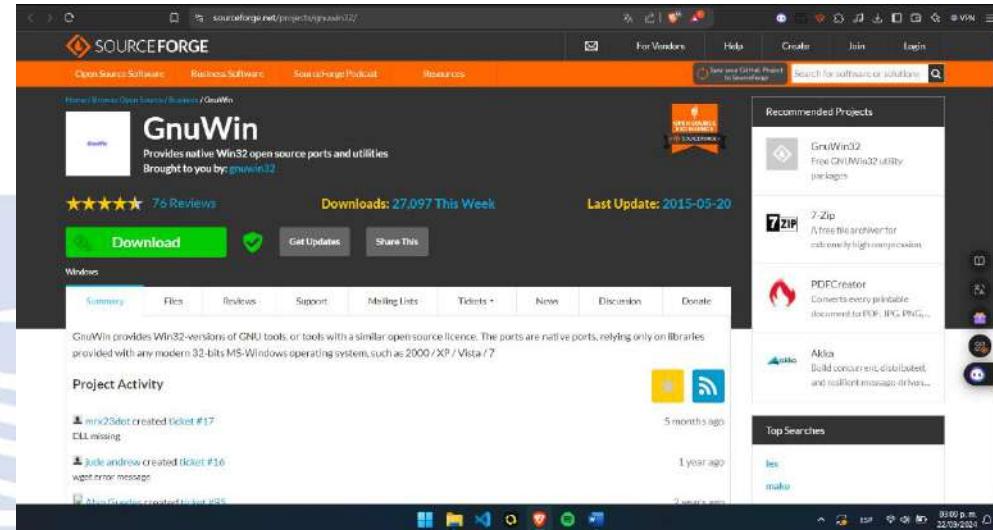


**TECNM**  
TECNOLOGICO NACIONAL DE  
**MÉXICO**

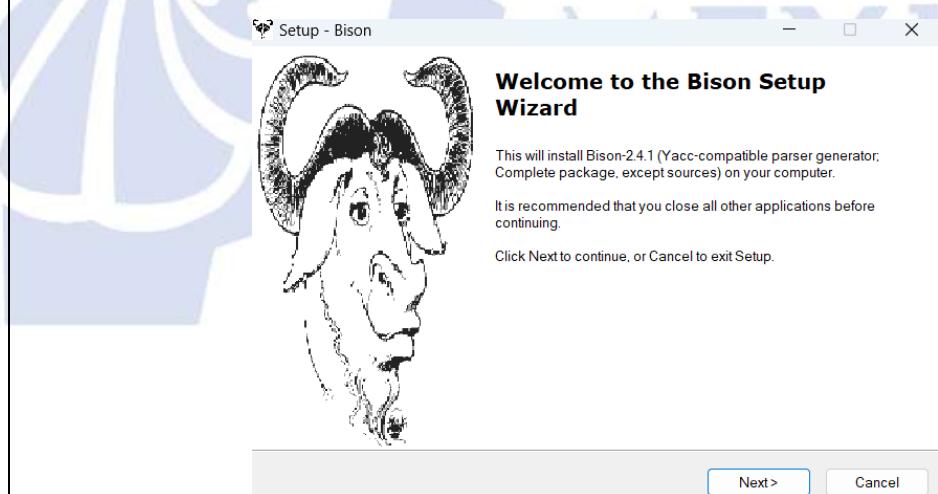
## ¿Cómo funciona el generador de código para analizadores sintácticos Bison?

Antes de comenzar con el ejemplo práctico de Bison debemos ver como se realiza su instalación. Dado que en prácticas previas ya se realizó la instalación de Flex el cual usaremos en conjunto con Bison para esta práctica, este tutorial será únicamente de como instalar Bison.

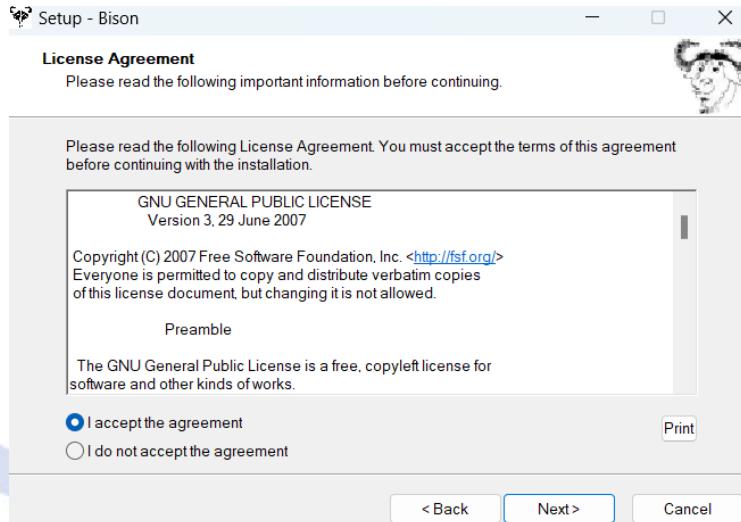
- Como primer paso se debe descargar el archivo de instalación desde la siguiente ruta  
<https://sourceforge.net/projects/gnuwin32/>



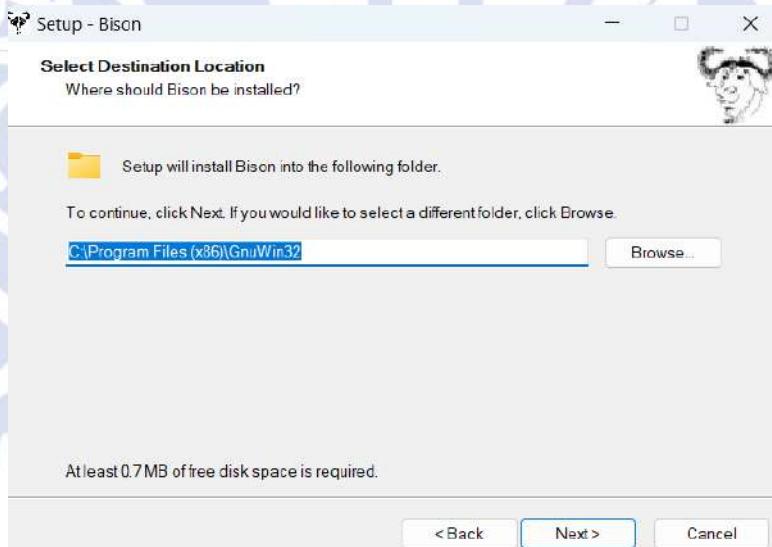
- Al ejecutar el instalador mostrar una pantalla como la siguiente, solo daremos clic en next.



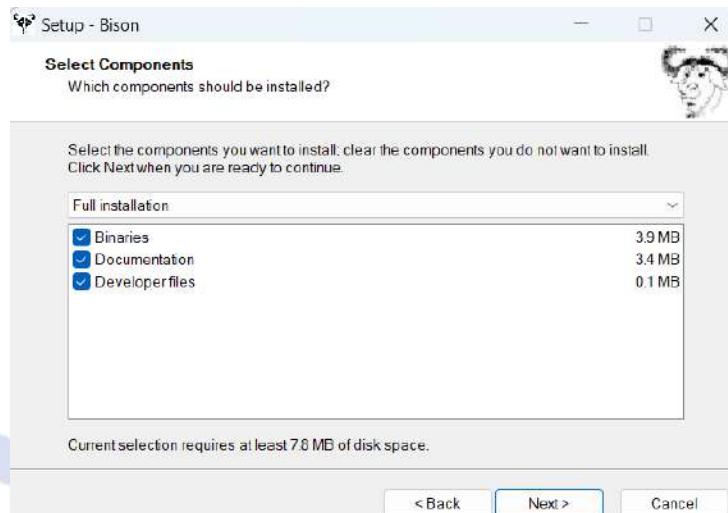
3. Aceptamos el acuerdo de licencia y damos clic en next.



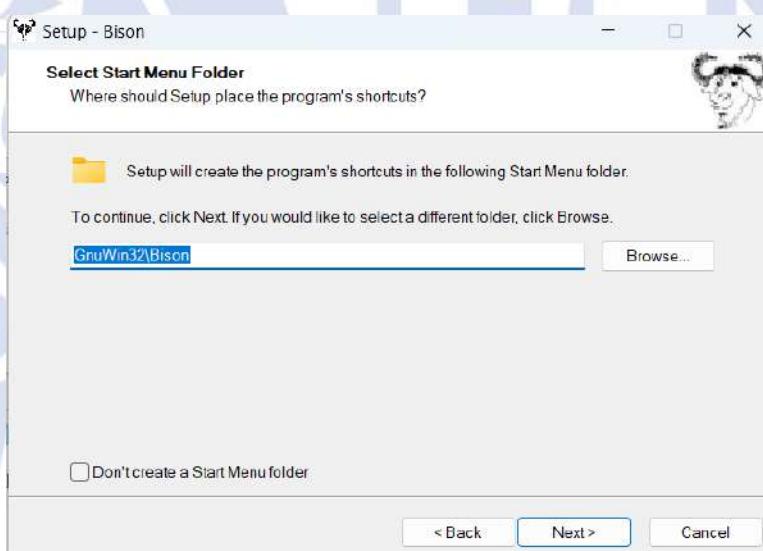
4. Se deja la ruta por defecto y hay que dar clic en next.



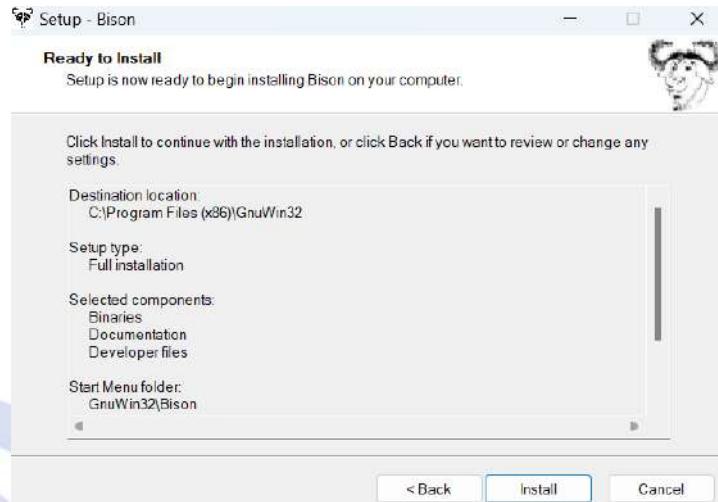
5. Una vez más solo hay que dar clic en next.



6. De nuevo solo daremos clic en next para crear el folder de la instalación.



7. La ultima pantalla solo mostrara un resumen de lo elegido para la instalación damos clic en install.



### Ejemplo práctico de Bison

Definir reglas léxicas con Flex.

1. Lo primero que se debe hacer es definir las reglas léxicas para identificar los tokens haciendo uso de Flex, aprovechando además el conocimiento previo en este. Para esta primera parte solamente se incluye y.tab.h que es un archivo generado por Bison que contiene las definiciones de los tokens, permitiendo que Flex conozca los nombres y valores de los tokens que Bison usa.

```
%{
#include "y.tab.h"
%}
```

2. En la definición de las reglas de Flex se definen las siguientes:

- [0-9]+ { return NUMBER; }: con esta expresión regular se detectaran secuencias de dígitos (números enteros), al encontrar un numero Flex retornara el token NUMBER y este será usado por Bison para manejar números.
- [ \t]+ ; : ignorara espacios en blanco y tabulaciones.
- \n { return '\n'; }: al detectar un salto de línea Bison usara el carácter '\n' para saber cuando se termine de ingresar una línea o expresión.
- "+" { return '+'; }: al encontrar un '+', se devuelve el mismo token '+', esto sirve en Bison para representar operadores aritméticos.
- "\*" { return '\*' }; igual que el caso anterior solo que retorna el token '\*'.
- "(" { return '('; } y ")" { return ')' }; detectan paréntesis de apertura y cierre.

```
%%
[0-9]+      { return NUMBER; }
[ \t]+      ;
\n          { return '\n'; }
"+"
{ return '+'; }
"**"
{ return '*'; }
"("
{ return '('; }
```

```
") "      { return ')'; }
```

```
%%
```

3. Por ultimo solo se tiene la función yywrap() que es llamada por Flex al ya no tener mas entrada que procesar, en este caso retornar 1 indica que debe terminar de leer la entrada.

```
int yywrap() {
    return 1;
}
```

#### Código de Flex completo

```
%{
#include "calculadora.tab.h"
}

%%

[0-9]+      { yyval = atoi(yytext); return NUMBER; } // Convierte el número en
entero y lo almacena en 'yyval'.
[ \t]+        ;
\n          { return '\n'; }
"+"         { return '+'; }
"**"        { return '*' ; }
"("          { return '('; }
")"          { return ')'; }

%%

int yywrap() {
    return 1;
}
```

#### Definir reglas sintácticas con Bison.

Este archivo contendrá las reglas sintácticas y el como se deben interpretar las secuencias de tokens que provienen de Flex. Bison se encargará de tomar esas secuencias de tokens para verificar que sigan la gramática definida para posteriormente ejecutar acciones asociadas a cada regla de la gramática.

```
%{
#include <stdio.h>
#include <stdlib.h>

void yyerror(const char *s);
int yylex(void);
%}
```

1. El primer bloque es similar a Flex, este es escrito en C y se incluyen funciones auxiliares que serán necesarias.
  - `#include <stdio.h>` y `#include <stdlib.h>`: son bibliotecas de C para la entrada/salida y gestión de memoria.
  - `void yyerror(const char *s);` : es una función usada para manejar errores sintácticos.

- int yylex(void); : esta función es la interfaz entre Bison y Flex. Bison la invoca para obtener el siguiente token de la entrada de Flex.

```
%token NUMBER
```

Aquí se declara el token NUMBER que representa un numero entero en el input y se retorna por Flex al detectar un número.

```
%%
input:
  | input line
  ;

line:
  '\n'
  | exp '\n' { printf("Resultado: %d\n", $1); }
  ;
```

## 2. El segundo bloque define las reglas sintácticas.

- Input: define que una entrada puede ser una secuencia de líneas (input line) o estar vacía.
- line: define que una línea puede ser un salto de línea, o una expresión seguida de un salto de línea. Si llega a ser una expresión, la acción que realizará será imprimir el resultado de la expresión con printf("Resultado: %d\n", \$1); , donde \$1 es el valor de la expresión calculada.

El bloque continuo de la siguiente manera:

```
exp:
  NUMBER { $$ = $1; }
  | exp '+' exp { $$ = $1 + $3; }
  | exp '*' exp { $$ = $1 * $3; }
  | '(' exp ')' { $$ = $2; }
  ;
```

- exp: define las posibles formas de una expresión aritmética. Bison toma los tokens generados por Flex y los combina según las reglas anteriores para formar expresiones validas.
- NUMBER { \$\$ = \$1; } : si se detecta que la expresión es un número, el valor de la expresión es el valor del número, \$1 es el valor del token NUMBER.
- exp '+' exp { \$\$ = \$1 + \$3; } : si la expresión es una suma de dos subexpresiones, la acción a realizar será la suma de los valores de las subexpresiones, \$1 y \$3 son valores de las subexpresiones a la izquierda y derecha de +.
- exp '\*' exp { \$\$ = \$1 \* \$3; } : si la expresión es una multiplicación de dos subexpresiones, se multiplica el valor de las mismas.
- '(' exp ')' { \$\$ = \$2; } : si la expresión está entre paréntesis, se ignoran los mismos y se devuelve el valor de la expresión interna.

```
%%
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main(void) {
```

```
    printf("Ingrese una expresión: \n");
    return yyparse();
}
```

3. Para el ultimo bloque se define lo siguiente:

- yyerror: función llamada cuando Bison detecta un error sintáctico, solo imprime el error en la consola.
- Main(): función principal con la que comienza el proceso de análisis. Comienza con un mensaje pidiendo una expresión por parte del usuario para posteriormente invocar yyparse(), la cual es una función generada por Bison para comenzar el análisis sintáctico.

Código de Bison completo.

```
%{
#include <stdio.h>
#include <stdlib.h>

void yyerror(const char *s);
int yylex(void);

%left '+'
%left '*'

%token NUMBER

%%
input:
    | input line
    ;

line:
    '\n'
    | exp '\n' { printf("Resultado: %d\n", $1); }
    ;

exp:
    NUMBER           { $$ = $1; }
    | exp '+' exp   { $$ = $1 + $3; }
    | exp '*' exp   { $$ = $1 * $3; }
    | '(' exp ')'
    ;
%%

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main(void) {
    printf("Ingrese una expresión: \n");
    return yyparse();
}
```

Ejecución de programa.

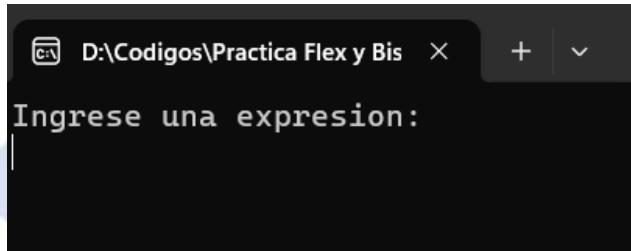
Mediante consola se ejecutan los siguientes dos comandos para generar los archivos de Flex y Bison.

```
D:\Códigos\Práctica Flex y Bison>flex calculadora.l  
D:\Códigos\Práctica Flex y Bison>bison -d calculadora.y
```

Posteriormente se ejecuta el siguiente comando para generar el archivo ejecutable.

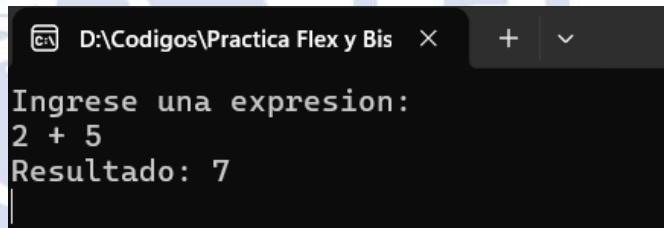
```
D:\Códigos\Práctica Flex y Bison>gcc calculadora.tab.c lex.yy.c -o calculadora -lfl
```

Al ejecutar el archivo .exe que se generó mostrara lo siguiente en consola:



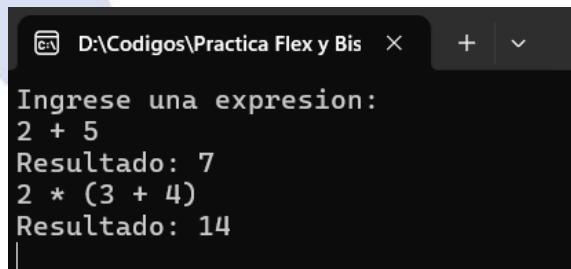
```
D:\Códigos\Práctica Flex y Bis > + v  
Ingrese una expresion:  
|
```

Como primera prueba se ingresará la expresión  $2 + 5$



```
D:\Códigos\Práctica Flex y Bis > + v  
Ingrese una expresion:  
2 + 5  
Resultado: 7  
|
```

Para la segunda prueba se usará una expresión como la siguiente:  $2 * (3 + 4)$



```
D:\Códigos\Práctica Flex y Bis > + v  
Ingrese una expresion:  
2 + 5  
Resultado: 7  
2 * (3 + 4)  
Resultado: 14  
|
```

### ¿Cuáles son las características y las funciones de estos analizadores sintácticos?

Yacc y Bison como ya se menciono en el apartado del marco teórico como durante la monografía que precede a esta practica son herramientas de software que generan código para analizadores sintácticos y son comúnmente usados en la construcción de compiladores.

Tienen como principal objetivo el traducir una gramática formal a un programa que pueda reconocer secuencias de tokens y validar que pertenezcan a la gramática.

Las principales características y funciones que presentan son las siguientes:

#### Yacc

Este fue desarrollado en los 70 siendo parte del sistema UNIX, además de ser uno de los primeros generadores de analizadores sintácticos.

##### Características:

- Basado en gramáticas LALR(1): Yacc se basa en la técnica Look-Ahead LR (LALR) para generar analizadores sintácticos permitiendo así el reconocer gramáticas libres de contexto.
- Integración con Lex: Yacc trabaja en conjunto con Lex, la cual es una herramienta para generar analizadores léxicos. Lex divide la entrada de tokens, mientras que Yacc los analiza sintácticamente.
- Personalización usando C: El código generado por Yacc es en lenguaje C, permitiendo la integración directa y capacidad de modificar el como se comporta el analizador sintáctico según lo que necesita el programador.
- Flexibilidad: Permite el definir acciones semánticas a ejecutar cuando una producción en específico de la gramática se reconoce.
- Fiable: Si bien Yacc ya ha sido reemplazado por herramientas mas modernas como el caso de Bison, sigue siendo confiable y ampliamente utilizado.

##### Funciones:

- Generación de analizadores sintácticos: se toma como entrada una gramática y produce un programa en C que analiza secuencias de tokens.
- Detección de errores sintácticos: puede manejar errores en la entrada y ofrece mecanismos para recuperarse de los mismos.
- Asignación de acciones semánticas: permite el asociar bloques de código a las reglas gramaticales, ejecutando acciones específicas al reconocer una producción.

#### Bison

Derivado de Yacc es desarrollado por el proyecto GNU, y si bien es compatible con Yacc presenta mejorar y es mas utilizado en la actualidad.

##### Características:

- Compatible con Yacc: Bison tiene compatibilidad con Yacc, es decir que puede procesar gramáticas escritas para Yacc sin tener que modificarla.
- Soporte y actualización: Bison es mas mantenido que Yacc al ser parte del proyecto GNU, asegurando soporte continuo y más actualizaciones.
- Soporte para lenguajes: Si bien Bison genera código en C por defecto, puede producir analizadores en C++, Java y más lenguajes.
- Análisis sintáctico avanzado: hace uso de la técnica LALR(1) pero tiene mejoras en el manejo de conflictos de gramática, recuperación de errores y eficiencia.
- Código abierto: es software libre y es parte del ecosistema de GNU.

##### Funciones:

- Generación de analizadores sintácticos: como Yacc, Bison genera un programa que

analiza la sintaxis de una entrada basada en una gramática dada.

- Resolución de conflictos: ofrece mecanismos para la resolución de conflictos entre reglas gramaticales de forma más eficiente.
- Manejo avanzado de errores: Proporciona mejor capacidad en la detección y recuperación de errores, dando robustez en el compilador.
- Soporte multilingüe: Aunque genera código en C, también puede generar código en lenguajes como C++, Java y más.

Como resumen se puede tener que si bien Yacc es la herramienta clásica y el primero para construir analizadores sintácticos, Bison ha sabido ser un buen sucesor moderno, ampliando las capacidades existentes en Yacc para ser mas utilizado que este mismo.



5

**BITÁCORA DE INCIDENCIAS**

PROBLEMA	FECHA	HORA	SOLUCIÓN	FECHA	HORA
No se podía ejecutar bison	21/09/2024	2:00 p.m.	Se tuvo que mover la carpeta de instalación	21/09/2024	2:30 p.m.
Problemas con la precedencia en Bison	21/09/2024	3:10 p.m.	Consultando videos y un ejemplo se entiende cómo aplicar precedencia correctamente.	21/09/2024	3:20 p.m.

6

**OBSERVACIONES**

Como ya se tenía previo conocimiento en el uso de Flex la parte donde este estaba involucrado fue relativamente sencilla, solo se uso para los tokens por lo cual fue fácil para el desarrollo del ejemplo práctico.

Por su parte el manejo de Bison llevo un poco mas de investigación, ya que aunque es un código fuente relativamente pequeño para el ejemplo usado durante esta práctica, tuvo sus complicaciones como la comentada en cuanto a la precedencia, fue necesario buscar un ejemplo que hiciera uso de esto para poder aplicarlo ya que el programa es una calculadora y no realizaba las operaciones, una vez aplicada correctamente la precedencia se soluciono este conflicto.

Además fue muy importante no usar una gramática ambigua en Bison ya que esto podría generar errores y una vez más se podía evitar con el uso de precedencia previamente mencionado.

7

**ANEXOS**

Un ejemplo muy importante para el desarrollo de esta practica fue el visto en el mismo video para la instalacion de Bison.

```

Code Blame 28 lines (20 loc) · 457 Bytes

1  %
2
3  #include <stdio.h>
4  #include <string.h>
5  #include "test.tab.h"
6  void showError();
7  %

8
9
10 numbers (([0-9])*)
11 alpha ([a-zA-Z])*
12
13 %%
14
15 {alpha}           {sscanf(yytext, "%s", yyval.name); return (STRING);}
16 {numbers}         {yyval.number = atoi(yytext); return (NUM);}
17 ";"               {return (SEMICOLON);}
18 .                 {showError(); return(OTHER);}

19 %%
20
21
22 void showError(){
23     printf("Other input");
24 }
25 int yywrap(){
26     return 1;
27 }

%{
#include <stdio.h>

int yylex();
int yyerror(char *s);

%
%token STRING NUM OTHER SEMICOLON

%type <name> STRING
%type <number> NUM

%union{
    char name[20];
    int number;
}

%%
prog:
    stmts
;

stmts:
    | stmt SEMICOLON stmts

stmt:

```

```

STRING {
    printf("Has ingresado un string - %s", $1);
}
| NUM {
    printf("El numero que ingresaste es - %d", $1);
}
| OTHER
;

%%
int yyerror(char *s)
{
    printf("Syntax Error on line %s\n", s);
    return 0;
}

int main()
{
    yyparse();
    return 0;
}
  
```

Con este ejemplo fue sencillo entender como se comunica Flex con Bison y como se hace el manejo de tokens.

Además una solución importante vino dada por el siguiente foro donde se logro solucionar el fallo de que bison no se ejecutara. <https://stackoverflow.com/questions/16442556/m4-no-such-file-or-directory-bison>

Moved the Folder from

 Explicar 

C:\Program Files(x86)\GnuWin32

to

C:\GnuWin32

then, added `PATH` values to both user variables and system variables as:

C:\GnuWin32\bin

- ProfeAntonio Ingeniería Sistemas e Informática. (2022, December 2). Analizador Sintáctico con FLEX Y BISON [Video]. YouTube. <https://www.youtube.com/watch?v=T1cptKHypQc>
- <https://stackoverflow.com/questions/16442556/m4-no-such-file-or-directory-bison>
- Rodríguez, J. M., & Pérez, J. F. (2008, enero 23). Generador de analizadores sintácticos BISON. CCIA, Universidade de Vigo. <https://ccia.esei.uvigo.es/docencia/PL/bison.pdf>
- Universitat Politècnica de València - UPV. (2011, November 29). Resolución de conflictos en

Bison | | UPV [Video]. YouTube. <https://www.youtube.com/watch?v=WQ54Du5lxA4>



## Videos

### Monografia

<https://youtu.be/4ZiR5hSCYWY>

### Practicas

<https://youtu.be/z5b1JDpDfwU>

<https://youtu.be/l84bcTkfSPg>

# ¿Por qué ASML tiene el monopolio más colosal de la historia?



23/09/2024

Lenguajes y Autómatas II  
Profesor: ISC. Ricardo González González

Instituto Tecnológico Nacional de México en Celaya

Presenta: Equipo No. 3

Isaac Salvador Bravo Estrada – 20030048

Guillermo Peasland Aguilar – 20030737

María del Carmen Chávez Patiño – 19030536

Luis Fernando Mendoza Javalera - 19030536

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

DOCENTE DESIGNADO
ISC. RICARDO GONZÁLEZ GONZÁLEZ

PRACTICA No.	NOMBRE DE LA PRACTICA	DURACIÓN ( HORAS )
8	ASML el monopolio mas colosal de la historia	2 horas

1

### **INTRODUCCIÓN**

En la industria de los semiconductores ASML se ha consolidado como el principal actor en cuanto a la fabricación de maquinaria para la producción de chips se trata. Durante la siguiente practica se analizará el cómo ASML alcanzo el éxito y estableció un monopolio colosal en la industria de la litografía, una tecnología esencial en la creación de chips. Superando a competidores como Nikon y Canon.

2

### **OBJETIVO ( COMPETENCIAS )**

Comprender y analizar el proceso que conlleva la creación de chips, la importancia de la litografía en la evolución de la industria de semiconductores y todo esto a través del estudio de ASML y sus innovaciones tecnológicas, identificando así factores clave que llevaron a la empresa alcanzar el éxito masivo en el mercado de semiconductores.

CE  
VW  
AS  
AF

**3**

### **MARCO TEÓRICO REFERENCIAL**

ASML es el líder mundial en tecnología de fabricación de semiconductores. Esta empresa proporciona herramientas y servicios de vanguardia para acelerar la adopción de nuevas tecnologías de fabricación de semiconductores. Y sus principales clientes son compañías como Intel, TSMC, Samsung, GF, etc., ya que sin ASML, estas otras empresas no podrían fabricar al nivel que lo hacen actualmente.

ASML, fundada en 1984, ha sido pionera en el desarrollo de tecnologías de litografía, un proceso que permite la impresión de circuitos en obleas de silicio mediante el uso de luz. La evolución de esta tecnología ha sido un pilar fundamental para la creación de chips más rápidos y eficientes. La litografía EUV, introducida por ASML, utiliza longitudes de onda extremadamente cortas (13.5 nm) para imprimir transistores miles de veces más pequeños que un cabello humano, lo que ha permitido avances significativos en la velocidad y capacidad de procesamiento de los chips.

Los semiconductores, materiales esenciales para el funcionamiento de chips son capaces de conducir o aislar electricidad según las condiciones, permitiendo así la creación de señales binarias dadas por 1 y 0. Los semiconductores dependen a su vez del uso de transistores para controlar el flujo eléctrico, y el punto mas clave en el avance de la velocidad de los chips a sido dado por la capacidad de miniaturizar los transistores.

**4**

### **MATERIALES UTILIZADO**

- Equipo de computo
- Navegador Web (Para consulta de información)

**5**

### **REQUISITOS BÁSICOS.**

- Sistema operativo: Windows 2000 / 98 / XP / Vista / 7 / 8 / 10 / 11
- Memoria (RAM): 256 MB
- Espacio disco duro: 100 MB

2023  
JULIA  
LIMA  
SANTOS

## DESARROLLO DE LA PRÁCTICA ( PASO A PASO )

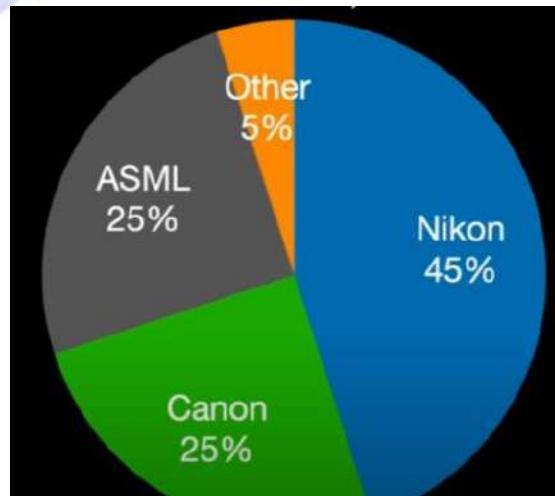
### ¿Por qué ASML tiene el monopolio mas colosal de la historia?

#### Caso de éxito

Todo dio inicio cuando un grupo de científicos se unen con un gran objetivo en mente “La litografía es el futuro de la fabricación de los chips”. Para el año 1984 se da la unión de Philips y ASM dando así el nacimiento de la compañía Advanced Semiconductor Material Lithography. Para el año 1985 se lanza la primera máquina que usaba tecnología de luz para imprimir diseños en silicio y así fabricar chips, esta tecnología lleva el nombre de FotoLitografía.



Los únicos que se subieron al barco de la investigación de la litografía además de ASML fueron dos grandes compañías, Nikon y Canon. Para el año 1996 el monopolio del mercado estaba ocupado por Nikon y Canon mientras que ASML se estaba quedando relegado.



ASML al ver que no podría competir en contra de Nikon y Canon, decidieron reunirse y emplear el poco dinero que les quedaba en una máquina de lo más innovadora, la TWINSCAN.



Esta maquina era capaz de realizar el proceso de medición e impresión de chips en la misma, acelerando así la fabricación de chips a 3 veces de la velocidad de otras maquinas de la época. Con esto Nikon y Canon se vieron incapaces de seguir el ritmo a ASML y dando así una vuelta en la mala racha que estaba teniendo la misma.

#### La idea

Uno de los factores clave si no es que el mas importante de la maquina de ASML es la luz, la luz es capaz de viajar en ondas electromagnéticas, las ondas pueden variar en frecuencia, y según la frecuencia es el color que somos capaces de percibir por parte de la luz así como la longitud de onda.



A mayor frecuencia menor longitud de onda, y ASML hizo posible crear una longitud de onda de 13.5 nm a la cual se le llama Extreme Ultraviolet (EUV) y esta es una longitud de onda 14 veces menos a la que se uso en su revolucionaria TWINSCAN.



APR 2024  
VITALE  
ZARDO  
HERRERA

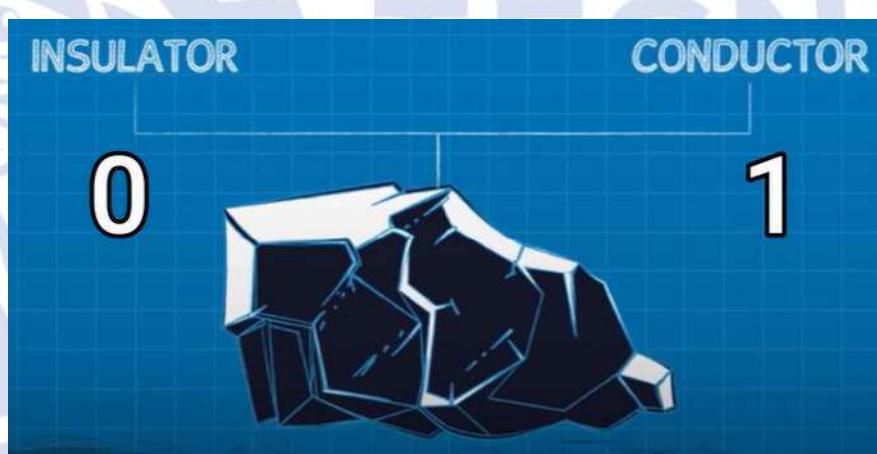
Con esto ASML logró que la impresión de chips tuviera transistores que son 10,000 veces más pequeños que un pelo humano, un verdadero avance en la creación de chips.

Si bien la idea de la máquina que usaba luz EUV para crear chips era muy innovadora, era igualmente arriesgado invertir tanto dinero en la idea, por lo cual la mejor solución que existía era el implicar a los principales clientes de ASML en el desarrollo mediante la venta de acciones de ASML, así asegurando el compromiso por parte de los clientes.

Y así con la creación de la máquina EUV se logra el monopolio por parte de ASML ahora nadie podría competir con ellos.

Los chips que hoy en día usa cualquier sistema funcionan en base de 1 y 0, todo son operaciones de 1 y 0 en los chips y esto se hace miles de veces por segundo, mientras más operaciones por segundo sea capaz de realizar un chip esto significa que será más rápido.

Y la manera de crear estos 1 y 0 es a través de semiconductores y electricidad. Los semiconductores son materiales que tienen la capacidad de conducir electricidad, pero también de poderla aislar, dando, así como resultado que generen esos 1 y 0 que se necesitan.



A su vez para controlar estas señales de 1 y 0 que se crean con los semiconductores se necesita de otro elemento importante, los transistores, estos cumplen la función de un interruptor el cual podrá dar paso o interrumpir la señal eléctrica según el camino que se quiera seguir.

Y retomando la parte de la velocidad de los chips, como ya se mencionó se necesita de poder realizar más operaciones de 1 y 0, siendo así que para tener mayor velocidad en un chip este debe tener más transistores por lo cual es necesario que estos sean cada vez más pequeños.

### Proceso de fabricación de un chip

Lo primero es recolectar arena, ya que esta tiene altas cantidades de silicio, el silicio es usado por ser muy abundante y también gracias a su estructura atómica, ya que si se le añade algún otro átomo es fácil convertirlo en un semiconductor.

Una vez purificado, se crea un lingote de silicio el cual posteriormente es cortado para forma un wafer.

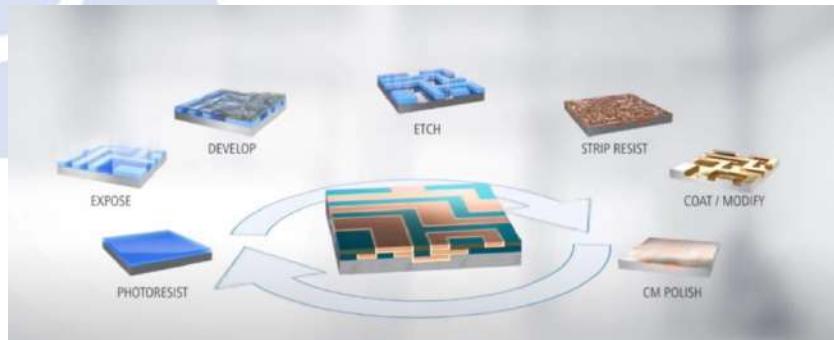
3  
4  
5  
6  
7



Sobre el wafer se depositan los materiales capa a capa empezando por una capa de resistencia a la luz para que al recibir la luz se endurezca, después la luz se pasa a través de un cristal con un diseño conocido como foto máscara y al pasar la luz a través de el wafer se ve como la luz deja caminos los cuales fueron diseñados por la foto máscara.

Ahora para la creación de la luz EUV que usa la maquina de ASML se empieza por lanzar unas gotas de estaño alrededor de 50 mil por segundo, posteriormente se les dispara un laser que impacta dos veces por gota generando así plasma que da como resultado la luz que viajara por los espejos y a través de la foto máscara que dejara impresos los caminos previamente mencionados.

Para finalizar se aplican capas de otros materiales, unos para eliminar material de áreas no expuestas a la luz, otras para ionizar el silicio, se añade una capa de metal para la circulación de la electricidad y finalmente se pule.



Con todo esto no es de extrañar que ASML haya creado un monopolio ya que son los únicos capaces de crear chips cada vez más rápidos.

Y aunque existen muchas compañías involucradas en este proceso de creaciones de chip ya sea vendiendo el material con el que se crea, otros que venden la maquinaria, otros que lo diseñan y los que venden los propios chips, al final del día solo es posible mejorar los chips si la maquina de

ASML es capaz de crear transistores más pequeños.

#### El mercado

Se podría pensar que cualquier otra compañía con el suficiente dinero podría competir con ASML, sin embargo, ellos tienen grandes contratos con proveedores y clientes, además que su tecnología avanza año con año. Y las compañías al tener contratos de exclusividad con ASML solo les permite el trabajar con la misma, es como un matrimonio sin divorcio.

A su vez en el mercado de semiconductores la demanda crece exponencialmente, todo mundo quiere mas chips, se necesitan para dar soluciones a cada vez más problemáticas, como la creciente IA, el auge de la nube y el internet de las cosas.

Y dado que la oferta se reduce ya que no muchas compañías fabrican chips, hace que ASML sea la única o mejor opción en cuanto a máquinas para la creación de chips se trata.



B. M. V. M. J. R.

5

### BITÁCORA DE INCIDENCIAS

Para el desarrollo de esta practica no se contaron con incidencias.

PROBLEMA	FECHA	HORA	SOLUCIÓN	FECHA	HORA

6

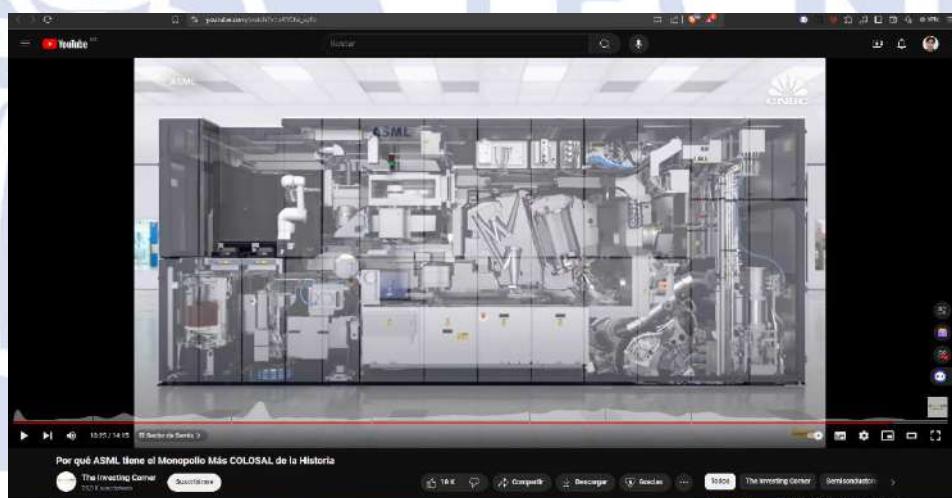
### OBSERVACIONES

- ASML es una compañía que supo ser visionaria en su momento y aun actualmente es una compañía que continúa innovando, si bien llegó a pasar por un momento de crisis y perder mercado en contra de Canon y Nikon, su perseverancia en la innovación los llevó a ser ahora la opción preferida por grandes compañías y generar un monopolio prácticamente, teniendo contratos de exclusividad y asegurando así su futuro en el mercado de chips.

7

### ANEXOS

El video proporcionado en la actividad fue la principal fuente de información usada para el desarrollo de esta práctica.



7

### REFERENCIAS BIBLIOGRÁFICAS

- The Investing Corner. (2023, September 18). Por qué ASML tiene el Monopolio Más COLOSAL de la Historia [Video]. YouTube. [https://www.youtube.com/watch?v=zAYCfw\\_syFc](https://www.youtube.com/watch?v=zAYCfw_syFc)
- Isaac. (2022, August 12). ASML: ¿Quiénes son y por qué son tan importantes? Profesional Review. <https://www.profesionalreview.com/2022/09/24/asml/>
- Timings, J. (2024, April 25). TWINS SCAN: 20 years of lithography innovation. ASML. <https://www.asml.com/en/news/stories/2021/twinscan-20-years-innovation>