

[Handwritten signatures]

INSTITUTO TECNOLÓGICO DE CELAYA

Lenguajes y Autómatas II

Equipo 2

Gomez Cabañas Anthony 20030057

Ortega Hernández Cristhian Alberto 20031091

Ponce Morales Alma Gabriela 20030274

Ruiz López Miguel Ángel 20030935

ACTIVIDAD 5 :

ANALIZADORES SINTÁCTICOS

I.S.C. Ricardo González González



DEPARTAMENTO DE SISTEMAS COMPUTACIONALES E INFORMÁTICA

ASUNTO: **SOLICITUD DE ACTIVIDADES**

Celaya, Guanajuato, 02 / octubre / 2022

LENGUAJES Y AUTÓMATAS II

DOCENTE DESIGNADO: ISC. RICARDO GONZÁLEZ GONZÁLEZ

SEMESTRE AGOSTO-DICIEMBRE 2023

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 :

- TEMA 3.3 PRECEDENCIA DE OPERADORES
TEMA 3.4.1 ANALIZADOR SINTÁCTICO DESCENDENTE (LL)
TEMA 3.4.2 ANALIZADOR SINTÁCTICO ASCENDENTE (LR, LALR)
TEMA 3.5 DISEÑO Y ADMINISTRACIÓN DE UNA TABLA DE SÍMBOLOS.
TEMA 3.6 MANEJO DE ERRORES SINTÁCTICOS Y SU RECUPERACIÓN.
TEMA 3.7 GENERADORES DE CÓDIGO PARA ANALIZADORES SINTÁCTICOS: YACC, BISON (REPASO).

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]

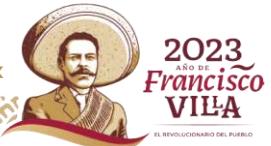
- B. ELABORE **UN VIDEO DE AL MENOS 25 MINUTOS**, DONDE A MODO DE VIDEO CONFERENCIA COMENTE Y EXPLIQUE LO DESARROLLADO TODOS LOS INCISOS ANTERIORES.

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.



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





CONSIDERACIONES.

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

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

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

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

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

OBSERVACIONES:

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

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

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





LA NOMENCLATURA SOLICITADA PARA ENVIAR SU TRABAJO ES LA SIGUIENTE :

AAAA-MM-
DD_TNM_CELAYA_MATERIA_DOCUMENTO_[EQUIPO]_NOCTROL_APELLIDOS_NOMBRE_SEM.PDF

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

DONDE :

TNM_CELAYA	:	INSTITUCIÓN ACADÉMICA
AAAA	:	AÑO
MM	:	MES
DD	:	DÍA
MATERIA	:	LAI _{II} , 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: AGO-DIC

EJEMPLO :

SI EL TRABAJO SE SOLICITÓ EN EQUIPO.

2023-10-02_TNM_CELAYA_LAI_{II}-A_A5_EQUIPO_99_9999999_PEREZ_PEREZ_JUAN_AGO-DIC23.PDF

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

SI EL TRABAJO SE SOLICITÓ INDIVIDUALMENTE.

2023-10-02_TNM_CELAYA_LAI_{II}-A_A5_9999999_PEREZ_PEREZ_JUAN_AGO-DIC23.PDF





FECHA Y HORA DE ENTREGA:

LA INDICADA EN LA PLATAFORMA VIRTUAL.

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

MUY IMPORTANTE:

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

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

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

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



[Handwritten signatures]

INSTITUTO TECNOLÓGICO DE CELAYA

Lenguajes y Autómatas II

Equipo 2

Gomez Cabañas Anthony 20030057

Ortega Hernández Cristhian Alberto 20031091

Ponce Morales Alma Gabriela 20030274

Ruiz López Miguel Ángel 20030935

Monografía:

Análisis
Sintáctico

I.S.C. Ricardo González González

VIDEO GENERAL: [Link](#)

ÍNDICE

Índice de recursos	2
Introducción	4
Generalidades	5
Desarrollo	6
1 Precedencia de operadores	6
2 Analizador sintáctico descendente (LL)	14
3 Analizador sintáctico ascendente (LR, LALR)	26
3.1 Algoritmo de desplazamiento-reducción (shift-reduce)	27
3.2 Gramáticas LR	29
3.3 Analizadores sintácticos ascendentes LR	30
3.3.1 Estructura y funcionamiento de un analizador	30
LR	
3.3.2 Análisis LRC(0)	32
3.3.3 Análisis SLRC(1)	33
3.3.4 Análisis LRC(1)	34
3.3.5 Análisis LALRC(1)	35
4 Diseño y administración de una tabla de símbolos	37
4.1 Acceso a la tabla de símbolos durante las distintas fases de un Compilador	40
4.2 Estructura de una Tabla de Símbolos	41
4.3 Administración de una tabla de Símbolos	42
5 Manejo de errores sintácticos y su recuperación	45
5.1 Información que deberá ofrecer un manejador de errores ante un error	48

~~A. J. M. G.~~

5.2 Estrategias de recuperación de errores en analizadores sintácticos	49
5.2.1 Corrección global	50
5.2.2 Modo Pánico	51
5.2.3 Modo de frase	52
5.2.4 Producciones de error	52
6 Generadores de código para analizadores Sintácticos	53
6.1 YACC	54
6.2 Bison	55
Conclusiones	57
Bibliografía	59

ÍNDICE DE RECURSOS

Figura	1 Procedencia de Operaciones	6
Figura	2 Tabla precedencia Java	7
Figura	3 Asociatividad de operadores	10
Figura	4 Expresiones aritméticas en gramática	11
Figura	5 Árbol Sintáctico para $2+3*4$	12
Figura	6 Ejemplo de recorrido del análisis sintáctico descendente	15
Figura	7 Construcción de la tabla	21
Figura	8 Funcionamiento básico del analizador LL	22
Figura	9 Gramática con prefijos comunes	27
Figura	10 Esquema analizador LR	31
Figura	11 AFD de elementos LR(0)	32
Figura	12 Acceso a la TS en las fases del compilador	40
Figura	13 Tabla de Símbolos globales y las de Alcance	44
Figura	14 Ejemplo impresión hello world java	45
Figura	15 Interacción Yacc/bison con proceso de análisis	53
Figura	16 Ejemplo archivo YACC	55

Introducción

En el marco de esta monografía, se pretende explorar a fondo el intrigante mundo de los analizadores sintácticos y su relevancia en el contexto de la compilación de programas. A lo largo de este documento, se abordarán diversos aspectos clave relacionados con la sintaxis de los lenguajes de programación y cómo se traducen en herramientas esenciales para la construcción de compiladores. Explorando temas como la precedencia de operadores y los diferentes tipos de analizadores sintácticos, incluyendo los analizadores sintácticos descendentes (LL) y ascendentes (LR, LALR). A su vez se abordaría en detalle los algoritmos de desplazamiento-reducción y las distintas categorías de gramáticos LR, como LR(0), SLR(1), LR(1) y LALR(1). Se analizará el acceso a la tabla de símbolos en las diferentes fases de un compilador, su estructura y su gestión eficiente. De esta manera se menciona la información necesaria que debe ofrecer un manejador de errores ante un error sintáctico. Dando también un vistazo a los generadores de código p. a analizadores sintácticos, incluyendo herramientas ampliamente utilizadas como Yacc y Bison.

GENERALIDADES

El propósito de esta monografía es explorar y analizar en profundidad varios temas fundamentales relacionados con la construcción de compiladores y procesadores de lenguajes. Cada uno de estos temas desempeña un papel crucial en el desarrollo de software y en la interpretación del código fuente, lo que los convierte en un conjunto de conceptos y técnicas esenciales.

En conjunto, estos temas son críticos para la comprensión, desarrollo y optimización de sistemas de análisis y procesamiento de código fuente.

1. PRECEDENCIA DE OPERADORES

A. M. G.

En programación, la precedencia de operadores se refiere a las reglas que establecen el orden en el que se deben ejecutar las operaciones de una expresión. Cuando una expresión tiene varios operadores, no todos tienen la misma prioridad.

La precedencia de operadores está estrechamente relacionada con las reglas de cálculo matemático, en muchos casos se suele utilizar las mismas reglas. Sin embargo, en los lenguajes de programación existen más operadores que se utilizan continuamente, lo que hace que las reglas de precedencia se vuelvan más complejas y extensas.

De manera general, la precedencia de operadores se sigue de la siguiente manera:

Precedencia	Categoría	Operadores
Más alta	Unario	+ - ! ~ ++ --
	Aritméticos	* / % + -
	Desplazamiento	<< >>
	Relacionales	< > <= >=
	Comprobación de tipos	is do
	Igualdad	= !=
	Lógicos	& ^
	Condicionales	&& ?:
Más baja	Asignación	= *= /= %= += -= <<= >>= &= = =

Figura 1. Precedencia de operadores

En un lenguaje de programación como Java, la precedencia de operadores es la siguiente:

Atenea

Atenea M. Gómez

Precedencia	Operador	Tipo
1	.	Asociación miembro
	[]	Arreglos
	()	Paréntesis
2	++	Unario post-incremento
	--	Unario post-decremento
	++	Unario pre-incremento
	--	Unario pre-decremento
	+	Unario suma
3	-	Unario resta
	!	Unario negación lógica
	~	Unario complemento
	(type)	Unario tipo objeto
	*	Multiplicación
4	/	División
	%	Módulo
5	+	Adición
	-	Restacción
	<<	Desplazamiento a la izquierda
6	>>	Desplazamiento a la derecha con extensión de signo
	>>>	Desplazamiento a la derecha con extensión 0 sin signo
	<	Relacional menor que
	<=	Relacional menor o igual que
7	>	Relacional mayor que
	>=	Relacional mayor o igual que
	instanceof	Tipo comprobación (solo objetos)
8	==	Relacional
	!=	Relacional es diferente a
9	&	AND
10	^	OR exclusivo
11		OR inclusivo

12	&&	AND lógico
13		OR lógico
14	? :	Escenario condicional
	=	Asignación
	+=	Adición asignación
15	-=	Restacción asignación
	*=	Multiplicación asignación
	/=	División asignación
	%=	Modulo asignación

Figura 2. Precedencia de operadores en Java

A continuación, se muestran ejemplos sobre la precedencia de operador:

En la expresión $10 + 5 * 2$, el operador $*$ tiene mayor precedencia que $+$, por lo que primero se realiza la operación $5 * 2$, y al resultado se le suma 10.

Los operadores de comparación tienen una precedencia más baja que los operadores aritméticos, es por eso que, en una expresión como $5 * 3 > 10$, primero se resuelve la operación $5 * 3$ y después se realiza la comparación. En este caso, el resultado sería verdadero.

En la expresión `true && false || true` se tienen los operadores $\&\&$ y $||$, que representan AND y OR respectivamente. AND tiene mayor precedencia que OR, por eso en la expresión se evalúa primero `true && false`, dando como resultado `false`; luego, se realiza la operación `false || true`, dando como resultado `true`.

1.1. UTILIZACIÓN DE PARÉNTESIS

Los paréntesis son utilizados para aislar la expresión de manera que el cálculo sea ejecutado de manera independiente. Permiten forzar a una expresión a ignorar las reglas de precedencia.

Por ejemplo:

En la expresión $1 + 3 * 4$ se respecta la precedencia de operador $*$, por lo que la expresión sería evaluada de la siguiente manera: $1 + (3 * 4) = 1 + 12 = 13$.

Si la expresión ahora es escrita de la siguiente forma $(1 + 3) * 4$, sería evaluada así: $(1 + 3) * 4 = 4 * 4 = 16$.

Como se mencionó anteriormente, los paréntesis permiten forzar que se realice una operación primero, lo que puede cambiar el resultado de las operaciones.

1.2. ASOCIATIVIDAD DE OPERADORES

La asociatividad de operadores es una propiedad que determina como se van a agrupar los operadores que tienen la misma precedencia cuando la expresión no tiene parentesis.

Cuando un operando está precedido y seguido por un operador, y los operadores tienen la misma precedencia, el operando puede ser utilizado como entrada para realizar dos operaciones diferentes. La elección de a qué operaciones aplicar el operando va a quedar determinada por la asociatividad.

~~A~~ ~~1~~ ~~2~~
A. M. G.

Los operadores tienen diferentes tipos de asociatividad:

- **Asociatividad no especificada:** La evaluación de los operadores va a depender de la implementación o las reglas del lenguaje. En otras palabras, los operadores pueden ser agrupados arbitrariamente.
- **Asociatividad por la izquierda:** Los operadores acá en evaluados de izquierda a derecha cuando tienen la misma precedencia.
- **Asociatividad por la derecha:** Los operadores acá en evaluados de derecha a izquierda cuando tienen la misma precedencia.

En la siguiente tabla se muestra la asociatividad de algunos operadores:

Tipo de operador	Operador	Asociatividad
Paréntesis	()	Izq. a dcr.
Exponentiación	^, **	Dcr. a izq.
Multiplicación y división	*, /, %	Izq. a dcr.
Suma y resta	+, -	Izq. a dcr.
Comparación	<, >, <=, >=, ==, !=	Izq. a dcr.
Lógicos	&&, , !	Izq. a dcr.
Asignación	=, +=, -=, *=, /=, %=	Dcr. a izq.
Terarios	? :	Dcr. a izq.

Figura 3. Asociatividad de operadores

1.3. AMBIGÜEDAD

Una gramática puede tener más de un árbol sintáctico que genera una cadena de terminal. Este tipo de gramáticas se dice que son ambiguas. Para realizar la demostración de que una gramática es ambigua, sólo se debe de encontrar una cadena de terminal que sea resultado de más de un árbol sintáctico. Debido a que una cadena con más de un árbol sintáctico generalmente tiene más de un significado, se debe diseñar una gramática que no sea ambigua para compilar aplicaciones o usar gramáticas ambiguas con reglas adicionales para resolver sus ambigüedades.

Las expresiones aritméticas tradicionales se pueden expresar mediante la siguiente gramática:

$$\begin{aligned} \text{Exp} &\rightarrow \text{Exp} + \text{Exp} \\ \text{Exp} &\rightarrow \text{Exp} - \text{Exp} \\ \text{Exp} &\rightarrow \text{Exp} * \text{Exp} \\ \text{Exp} &\rightarrow \text{Exp} / \text{Exp} \\ \text{Exp} &\rightarrow \text{num} \\ \text{Exp} &\rightarrow (\text{Exp}) \end{aligned}$$

Figura 1. Gramática de expresiones aritméticas

Lo importante notar que "num" es un terminal, que denota todos los constantes enteras y, además, los paréntesis son símbolos terminales.

Esta gramática es ambigua, como se puede notar en la producción $\text{Exp} \rightarrow \text{Exp} + \text{Exp}$. Para comprender mejor esta ambigüedad, veamos la siguiente expresión: $2 + 3 * 4$.

~~A. M. G.~~

Cada expresión puede ser leída de dos maneras:

- Sumar primero $2 + 3$, y al resultado multiplicarlo por 9 .
- Multiplicar 3×9 , y al resultado sumarle 2 .

Las calculadoras comunes utilizan la primera interpretación, ya que estos realizan el cálculo leyendo la expresión de izquierda a derecha; en cambio, las calculadoras científicas y la mayoría de los lenguajes de programación utilizan la segunda interpretación, ya que utilizan la jerarquía indicada por la precedencia de operadores.

Idealmente, la expresión $2+3*9$ debería generar el siguiente árbol sintáctico:

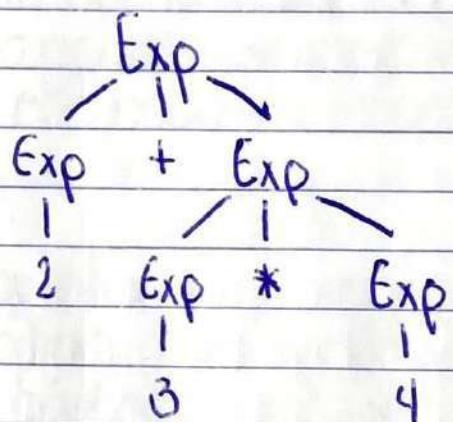


Figura 5. Árbol sintáctico para la expresión $2+3*9$

En ese árbol se reflejan las precedencias de operadores agrupándolos en subexpresiones. Cuando se evalúa la expresión, las subexpresiones que quedan representadas en el árbol sintáctico son evaluadas dentro de qué se aplique el operador con mayor prioridad.

~~A. M. S.~~

Una manera de aclarar esta ambigüedad es usar las reglas de precedencia durante el análisis sintáctico para seleccionar entre los posibles árboles sintáticos. La mayoría de generadores de analizadores sintáticos permiten este agrupamiento, sin embargo, algunos métodos de análisis sintáctico requieren que las gramáticas no tengan ambigüedad, es decir que se tiene que expresar la jerarquía de operabilidad en la misma gramática.

2. Analizar sintáctico descendente (LL)

2.2 Análisis sintáctico descendente

Un analizador sintáctico descendente, también llamado "parse top-down", desempeña un papel esencial en un compilador o procesador de lenguaje. Su tarea principal consiste en evaluar si una secuencia de tokens proporcionada previamente por el analizador léxico, puede conformar una instrucción válida en un lenguaje particular, ya sea un lenguaje de programación o un lenguaje natural.

Este analizador lleva a cabo la construcción de un árbol sintáctico basado en los tokens proporcionados, en el cual estos tokens se convierten en las hojas del árbol. El proceso de construcción de la estructura jerárquica de la sentencia comienza desde la raíz y se extiende hacia las hojas del árbol sintáctico. En el caso de que el analizador sintáctico no puede generar una sentencia válida, emite un mensaje de error que señala razones del problema, lo que facilita la comprensión y corrección de código.

Gramática

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E^n E \\ E &\rightarrow id \\ E &\rightarrow \text{num} \end{aligned}$$

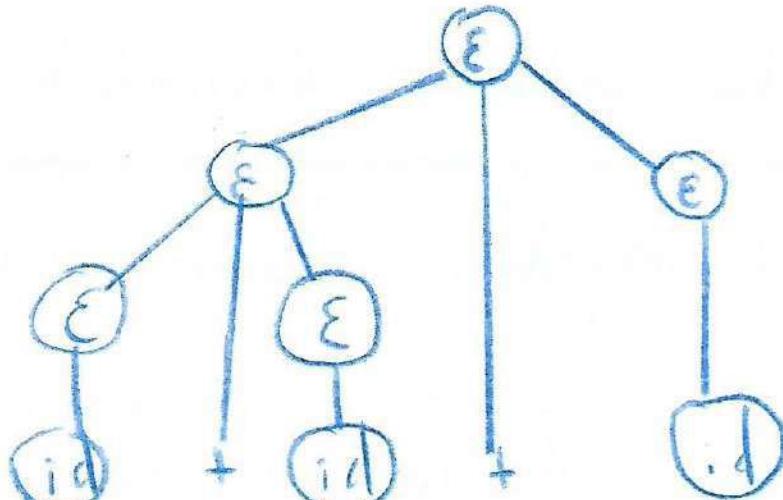


Figura 6 Ejemplo de recorrido del análisis sintáctico descendente

El procedimiento de análisis descendente se caracteriza como un proceso que establece metas y submetas al intentar relacionar una oración con su estructura sintáctica. Las submetas son examinadas y eliminadas, y la secuencia continua hasta que se logra la meta final, que son los símbolos terminales que componen la oración.

En el pasado, los compiladores orientados a la sintaxis empleaban el análisis recursivo descendente, pero se enfrentaron a desafíos significativos, como la necesidad de retroceder en el proceso, lo que limitó su utilidad práctica. Sin embargo, con la aparición de las gramáticas LL(1) y posteriormente las gramáticas LL(k), se logró realizar el análisis sin necesidad de retroceso.

2.2 El retroceso

El desafío del retroceso en el análisis sintáctico descendente se presenta cuando el analizador sintáctico, al analizar una sentencia, se encuentra en la situación de tener que "retrroceder": o regresar en su proceso de análisis debido a dificultades para determinar la estructura sintáctica correcta.

Esta situación surge cuando el analizador ha tomado una decisión inicial sobre cómo interpretar la sentencia, pero luego se da cuenta de que esta elección es incorrecta y necesita revisar su enfoque.

El retroceso puede ser necesario por diversas razones, que incluyen la ambigüedad gramatical, en la cual una gramática permite múltiples interpretaciones de una sentencia, lo que puede llevar a una elección inicial incorrecta. También puede deberse a reglas de producción no deterministas que no proporcionan suficiente información para decisiones sin ambigüedades, o a casos en los que el contexto no está claro y se requiere información adicional.

antes de tomar una decisión.

Este problema del retroceso puede tener implicaciones en términos de rendimiento y complejidad del analizador sintáctico. Para abordar este desafío, se han desarrollado estrategias como la utilización de gramáticas LL(1) y el análisis sintáctico predictivo. Estas técnicas permiten tomar decisiones sin necesidad de retroceder, al examinar un número limitado de símbolos hacia adelante en la entrada, lo que conduce a una mejora en la eficiencia y la previsibilidad del análisis sintáctico descendente.

2.2.1 Ejemplo

Para que lo anterior quede más claro es necesario ilustrarlo con un ejemplo matemático como el que se puede ver a continuación:

$$2 + 3 * 5$$

Se encuentra con un analizador sintáctico descendente y se encuentra como obstáculo una gramática ambigua.

Para el análisis se encuentra el número "2". Se asume que se ha identificado una expresión válida y se continua con la evaluación.

A continuación, se encuentra el operador de suma ($+$). Siguiendo la suposición anterior, concluimos que estamos calculando una suma y avanzamos en el análisis.

Luego se topa con el número 3 , lo que refuerza nuestra creencia de que estamos realizando $2+3$. En consecuencia, se avanza aún más en la expresión.

Sin embargo, la situación se complica cuando se encuentra el operador de multiplicación ($*$). Este operador nos indica que se trata de una multiplicación, lo que contradice la suposición anterior de una suma. En este punto, nos damos de que se ha cometido un error y que necesita reconsiderar la interpretación.

Para corregir esta equivocación, se debe retroceder y volver a evaluar la expresión como $2 + (3 * 5)$. Esto se debe a que la multiplicación tiene una prioridad superior en las operaciones matemáticas, y se necesita reflejar esta prioridad en el análisis.

~~Algunas claves~~
El proceso de retroceso implica que el analizador sintático debe revisar y corregir su decisión anterior para construir adecuadamente el árbol sintáctico de la expresión. Este ejemplo ilustra cómo el retroceso puede ser necesario cuando se enfrenta a gramáticas ambiguas o expresiones con reglas de precedencia de operadores. Una forma de evitar esta dificultad es diseñar gramáticas que sean más deterministas y no ambiguas, lo que simplifica el proceso de análisis sintáctico.

2.3. Construcción de la tabla de análisis LL.

La tabla de análisis sintáctico $LL(1)$ constituye una estructura de datos bidimensional que se organiza mediante una indexación basada en dos categorías principales: los terminales (T), que forman la fila superior de la misma, incluyendo el símbolo de fin de entrada ' $\$$ ', y los no terminales (N), que se encuentran en la columna izquierda de la tabla. Se puede acceder a esta tabla por medio de la notación $M(N, T)$.

Para abordar los errores en el análisis sintáctico, en una etapa posterior para tratar errores, se pueden

llenar las celdas vacías en la tabla, las celdas vacías indican la posibilidad de errores en el análisis sintáctico.

llenar las celdas vacías mediante llamadas a funciones o procedimientos que notifican y gestionan los errores detectados durante el proceso de análisis sintáctico.



~~Añadir A_t~~ Se añade $A \rightarrow a$ en $M(A, t)$ para cada terminal de PRIMERO(a)



S: la cadena está vacía en PRIMERO
se añade $A \rightarrow \lambda$, también aplica para S

Figura 7 Construcción de la tabla

2.4. Funcionamiento básico del analizador LL.

El analizador sintáctico descendente LL(1) emplea una pila como parte de su proceso de análisis sintáctico, siguiendo una arquitectura inspirada en la de un autómata de pila. Además, se vale de una tabla de análisis que determina si la entrada es válida y qué acción debe tomar en función de la entrada. A continuación se puede observar una figura que representa el funcionamiento básico:

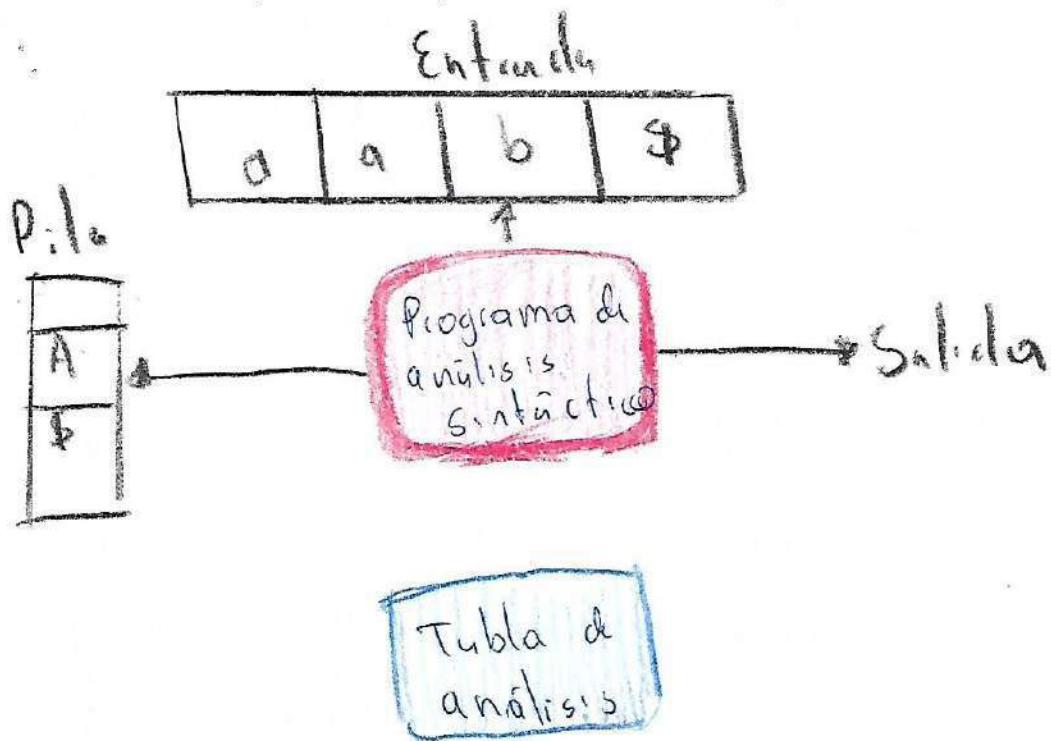


Figura 8 Funcionamiento básico del analizador LL

El proceso del analizador es determinar si el
lo proporcionado pertenece a la gramática de
finida. Se logra utilizando una pila para al-
macenar varios símbolos gramaticales, abarcán-
do terminales y no terminales, comenzando con el
símbolo ' $\$$ ' al principio de la pila y al
final de la cadena de entrada. De esta ma-
nera se asegura que, si no hay errores de-
tectados y se detecta al final ' $\$$ ', se con-
sidera como válida según la gramática y se
concluye el análisis con éxito. El programa
de análisis consulta la tabla de análisis pa-
ra determinar qué producción gramatical aplicar
en función de los símbolos presentes en la pila
y la entrada. Entonces se puede decir que el
proceso comienza marcando el fondo de la
pila y colocando en ella el símbolo inicial
de la gramática. Luego se aplica el siguiente
conjunto de reglas:

2.5 Aplicaciones

Los analizadores sintácticos descendentes LL(2) encuentran aplicaciones diversas en el ámbito de la programación y el procesamiento de lenguajes.

Entre los más destacados se incluyen:

- Análisis de lenguaje natural: Se aplican en el procesamiento de lenguaje natural para analizar y comprender la estructura sintáctica de las frases y textos escritos en lenguaje natural, lo que permite identificar componentes gramaticales en las oraciones, como sujetos, etc. y resulta útil en tareas como la traducción automática.

Compiladores: Permiten examinar la estructura sintáctica del código y generar un árbol de derivación que representa la gramática subyacente del programa.

- Verificación de gramáticas: Facilitan la comprobación de si una gramática es de tipo LL(2), es decir, si puede ser analizada.

~~IDEs~~ ~~IDEs~~ IDEs y editores de código: Estas herramientas utilizan los analizadores sintácticos descendentes para analizar el código ingresado por los desarrolladores y proporcionan tanto sugerencias como correcciones.

3. ANALIZADOR SINTÁCTICO ASCENDENTE

Un análisis sintáctico ascendente se refiere a la construcción de un árbol de análisis para una cadena de entrada impuesta por las hojas (la parte inferior del árbol) y avanzando hacia la raíz (la parte superior del árbol). El analizador sintáctico se puede describir como el proceso de construir árboles de análisis, aunque se podría llevar a cabo una traducción sin construir explícitamente un árbol.

Desde un punto de vista conceptual, un analizador ascendente opera al mantener un conjunto de subárboles parcialmente construidos que componen el árbol de análisis sintáctico. Estos subárboles se combinan cada vez que el analizador reconoce los símbolos del lado derecho de alguna producción utilizada en la parte más cercana de la cadena de entrada. En este proceso, se genera un nuevo nodo interno y, así mismo, como hijos de ese nodo las raíces de los subárboles que se están uniendo.

En la práctica, un analizador ascendente utiliza principalmente una pila de análisis. Este tipo de analizador mantiene las raíces de los subárboles parcialmente construidos en una pila. Cuando recibe un nuevo símbolo del analizador léxico, lo agrega a la pila. Cuando identifica que los símbolos en la parte superior de la pila conforman el lado derecho de una producción gramatical, lo reduce al lado izquierdo de esa producción, retirándolos de la pila y colocando el lado izquierdo en su lugar. La función de la pila es la primera diferencia clave entre el analizador ascendente y el descendente: en un analizador descendente, la pila contiene una lista de lo que el analizador anticipa encontrar en el futuro; en un analizador ascendente, la pila registrará lo que el analizador ha

procesado en el parádo.

Debido a su potencia, eficiencia y facilidad de construcción los analizadores ascendentes se utilizan comúnmente en la etapa de verificación de la sintaxis en un compilador. Los características gramaticales que pueden plantear problemas en el análisis descendente, como producción recursiva a la izquierda y prefijos comunes, suelen ser manejados de manera más efectiva en el análisis ascendente. Por ejemplo, la siguiente gramática

$\text{Sstmt} \rightarrow \text{if Expr then SstmtList endif} \mid \text{if Expr then}$

$\text{SstmtList} \rightarrow \text{SstmtList} \text{ else SstmtList endif}$

$\text{SstmtList} \rightarrow \text{SstmtList; Sstmt} \mid \text{Sstmt}$

$\text{Expr} \rightarrow \text{var} + \text{Expr} \mid \text{var}$

Figura 9. Gramática con prefijos comunes

es lo suficientemente clara como para definir la sintaxis de un lenguaje. No obstante, debido a los prefijos comunes y las reglas recursivas a la izquierda, esta gramática no resulta adecuada para el análisis descendente.

3.1 ALGORITMO DE DESPLAZAMIENTO-REDUCCIÓN (SHIFT-REDUCE)

Los analizadores sintácticos ascendentes utilizan un algoritmo llamado "desplazamiento-reducción" para poder construir el árbol de análisis sintáctico.

De manera general, este algoritmo permite analizar la sintaxis de una entrada mediante la construcción de un árbol de análisis sintáctico en medida que se procesan los tokens.

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

Cada "desplazamiento" agrega un token a la pila, y cada "reducción" reemplaza símbolos de la pila con producciones de la gramática. Este proceso se repite hasta que se detecta el círculo o se encuentra un error.

A continuación, se explora con más detalle cómo funciona este algoritmo:

1.- **Inicialización:** Se inicia con una pila vacía y una entrada que contiene la secuencia de tokens que se deseán analizar. A la pila, como primer paso, se le añade un símbolo especial llamado "estado inicial".

2.- **Iteración:** Este algoritmo opera en un bucle hasta que se haya analizado completamente la entrada y se haya construido el círculo de análisis sintáctico o hasta que se detecte un error.

3.- **Desplazamiento:** En cada iteración, se examina el símbolo que se encuentra en la parte superior de la pila (estado actual), y el siguiente token de la entrada. Si existe una transición válida en la gramática que va desde el estado actual al siguiente token de entrada, se realiza un desplazamiento; el token de entrada se coloca en la parte superior de la pila como un nuevo estado, y se avanza al siguiente token de entrada en la secuencia.

4.- **Reducción:** Si en algún punto se identifica que los símbolos en la parte superior de la pila representan una producción válida de la gramática, se realiza una operación de reducción. Durante esta operación, se retiran los símbolos de la parte superior de la pila que corresponden al lado derecho de una

producción gramatical válida y se reemplazan por el símbolo no terminal correspondiente al lado izquierdo de cada producción. Esto refleja la reducción de varios símbolos en la pila a un solo símbolo no-terminal.

5.- Repetición: Se continúan los pasos de desplazamiento y reducción alternativamente hasta que se haya construido todo la entrada y se haya construido un árbol de análisis sintáctico completo, o hasta que se detecte un error.

6.- Finalización: Si el análisis tiene éxito, se ha construido un árbol de análisis sintáctico, el proceso termina; si se detecta un error, se informa al usuario sobre la ubicación y naturaleza del error en la entrada.

3.2. GRAMÁTICAS LR

Los analizadores sintácticos asociales utilizan métodos de análisis LR, lo que les permite reconocer la mayoría de las construcciones de los lenguajes de programación. Estos métodos son más potentes y tienen un mayor poder de reconocimiento que los métodos LL.

Para el uso de estos métodos, es importante contar con gramáticas LR(k). Este tipo de gramáticas se caracteriza por:

- Procesar la cabecera de entrada de izquierda a derecha (left-to-right).
- Proporcionar la derivación más a la derecha de la cadena de entrada en orden inverso (Right-most derivation).
- Examinar k-símbolos de la entrada por anticipado para poder tomar una decisión sobre la acción que se debe de realizar.

Para que una gramática sea sea LR(k), es decir que poderse conocer la procedencia del lado derecho de una producción dso k -es, de haber visto todo lo que deriva del lado derecho y usando k -simbolos por anticipado.

3.3. ANALIZADORES SINTÁCTICOS ASCENDENTES LR

Los analizadores que se utilizan en la compilación y análisis sintáctico de lenguajes de programación. Son conocidos por ser muy sencillos y capaces de analizar una amplia variedad de gramáticas, incluso las que contienen producciones recurriendo a la ignorada y ambigüedad.

- (1) El término LR hace referencia a la manera en como se construye el árbol de análisis sintáctico mientras se procesa la entrada.
- (2) analizador trabaja de izquierda a derecha, leyendo los símbolos de entrada en ese orden, y construye el árbol de análisis sintáctico utilizando derivación hacia la derecha.

Existen varios tipos de analizadores LR, como LR(0), SLR(1), LALR(1), LR(1), que difieren en su capacidad para manejar diferentes tipos de gramáticas y la cantidad de información de anticipación que consideran al tomar decisiones.

3.3.1. ESTRUCTURA Y FUNCIONAMIENTO DE UN ANALIZADOR LR

Un analizador LR se compone de tres partes principales: un programa de análisis LR, una tabla de análisis y una pila. El programa de análisis LR es el responsable de llevar a cabo el análisis sintáctico. La tabla de análisis especifica las acciones que el analizador debe tomar en función del estado actual y los símbolos de entrada. La pila es usada para registrar los estados por los que pasa el analizador y los símbolos de

la gramática que se está leyendo. El analizador toma una entrada y la procesa, produciendo una salida como resultado del análisis sintáctico.

Lo único que varía de un analizador LR a otro es la tabla de auxiliares, que se compone de dos partes: una función llamada "acción" que determina las acciones a tomar, es decir, qué ha de hacer a continuación, y otra función llamada "goto" que decide a qué estado debe trasladarse el analizador. Todo lo demás en los analizadores LR es idéntico. El programa de auxiliares LR tiene carácter uno a uno desde un buffer de entrada. Utiliza una pila para almacenar una cadena en la forma $\lambda_0\lambda_1\lambda_2\lambda_3\dots$. Además, donde λ_m es el elemento en la parte superior de la pila.

Cada λ_i representa un símbolo de la gramática, y cada λ_i es un símbolo que representa un estado. Los símbolos de estado contienen toda la información contenida en la pila por debajo de él. La combinación del contenido en la parte superior de la pila y el símbolo de entrada actual se utilizan como índice en la tabla de auxiliares para determinar la acción siguiente que debe realizar el analizador.

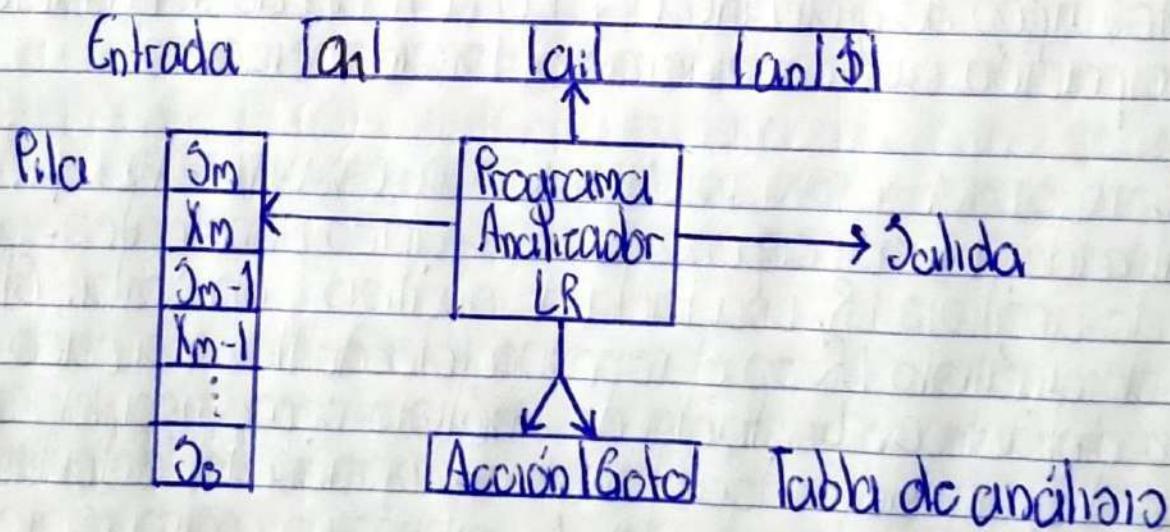


Figura 10. Esquema de un analizador LR

[Handwritten signatures]

El símbolo \$ indica el final de la cadena de entrada. El programa de análisis determina el contenido en la parte superior de la pila y el símbolo de la entrada que debe leer. Luego, consulta la acción correspondiente en la tabla de auxiliares bajo la entrada "acción (\$, q1)", donde \$m representa el contenido actual y q1 es el símbolo de entrada. Las acciones posibles pueden ser las siguientes:

1. **regar**: agregar un contenido S a la pila.
2. **Reducir**: usando una regla de producción de la gramática.
3. **Aceptar**: indicando que se ha completado el auxiliar.
4. **Error**: señalando que se ha encontrado un error de sintaxis.

La función `acto` toma un estado y un símbolo de la gramática como argumentos y produce un nuevo estado.

3.3. 2. Análisis LR(0)

Este es el más sencillo de todos los métodos ascendentes, ya que este no usa información de la entrada, el símbolo de precedencia, para decidir la acción a realizar.

El auxiliar LR(0) se basa en la idea de que una reducción puede ocurrir cuando el punto esté al final de una producción o; contenido y se encuentra un símbolo de终结符 valido. Esto significa que se puede reducir la secuencia de símbolos en el contenido a un símbolo no terminal correspondiente a la producción.

Este auxiliar puede presentar conflictos en ciertas gramáticas, como conflictos de desplazamiento-reducción o reducción-reducción. Estos conflictos ocurren cuando hay ambigüedad en la gramática.

3.3.2.1. ALGORITMO

Este algoritmo no requiere de ningún token de precancillado.

Cuando se alcanza un estado en el que sólo se pueden realizar acciones de reducción, se siguen aplicando sin necesidad de verificar el siguiente token de la cadena de entrada.

Cuando se alcanza un estado que no es de reducción, el token de la cadena de entrada es leído y se ejecuta la acción de desplazamiento. Es posible que se llegue a un estado de aceptación o error.

3.3.2.2. LIMITACIONES

- No puede manejar gramáticas ambiguas
- Existen conflictos de desplazamiento-reducción y reducción-reducción
- El autómata que se construye es muy grande y complejo
- Puede tener dificultades para detectar algunos errores sintácticos.

3.3.3. ANÁLISIS SLR(1)

Utiliza el autómata finito determinista que se construye a partir de los elementos LR(0) y utiliza el token de la cadena de entrada para determinar qué acción realizar.

3.3.3.1. ALGORITMO

Si el símbolo de entrada coincide con el que sale en la parte superior de la pila, se desplaza a la pila y se avanza al siguiente símbolo de entrada.

Si el símbolo que está en la parte superior de la pila coincide con una estructura gramatical, se reduce esa estructura a un solo símbolo.

Si no se puede realizar ninguna acción, se encuentra un error de sintaxis.

3.3.3.2. LIMITACIONES

Al igual que el analisis anterior, hay conflictos de desplazamiento-reducción y reducción-reducción, pudiendo ocurrir automáticamente más grande y tiene dificultades para detectar errores sintácticos.

Aunque todavía existen conflictos de desplazamiento-reducción y reducción-reducción, estos no suelen ser habituales y pueden ocurrir debido a un mal diseño de la gramática.

3.3.4. ANÁLISIS LR(1)

Este es el método más general que existe, además de ser el más popular. Una gran desventaja que tiene es que se usa mucho más tiempo que cualquier otro método, lo que hace que requiera más tiempo y memoria para construir y recorrer el automático de estados.

Gracias a su gran poder, se pueden manejar gramáticas más extensivas, así como ambiguas que otros métodos no pueden.

3.3.4.1. Algoritmo

Para realizar una reducción, se verifica que el estado actual de la pila coincida con una regla de producción completa, es decir, una regla de la gramática donde ya se han reconocido todos los símbolos necesarios, y el siguiente símbolo de la

[Handwritten signatures]

entrada es uno especial, entonces se puede aplicar esa regla de producción. Cuando se cumplen esas condiciones, se eliminan los símbolos de la pila que corresponden a la regla de producción y se reemplazan por un símbolo no-terminal que representa el resultado de la regla.

Para realizar un desplazamiento, se verifica que el carácter actual de la pila coincida con el símbolo que coincide con el siguiente símbolo de la entrada. Si coinciden, se toma el símbolo de la entrada y se coloca en la pila.

3.3.4.2. LIMITACIONES

Este tipo de análisis son muy grandes, por lo tanto, puede requerir una cantidad significativa de memoria para almacenar y generar, lo que hace que el proceso de análisis sea lento y se necesiten muchos recursos. Además, la construcción de las tablas puede ser complicada y estar propensa a errores.

3.3.5. ANÁLISIS LALR(1)

Este es una extensión del LR(1) cuyo objetivo es superar algunas de las limitaciones de ese análisis. En el análisis LALR(1), se construye un automata de estados similar al del análisis LR(1), pero se fusionan ciertos estados para reducir la complejidad y el tamaño del automata. Esto se logra al agrupar estados que tienen conjuntos de elementos LR(1) similares. La fusión de estados en el análisis LALR(1) puede generar una tabla de análisis más compacta y eficiente.

Aunque el análisis LALR(1) es menos poderoso que el análisis LR(1) en términos de reconocimiento de gramáticas, sigue siendo capaz de reconocer una amplia gama de gramáticas.

usadas en la mayoría de los lenguajes de programación.

3.3.5.1. ALGORITMO

Este método opera de manera que el algoritmo visto en el método LR(1).

3.3.5.2. LIMITACIONES

Es más expensivo que el δ LR(1), pero aún así no puede manejar todos los ambiguos de gramáticas.

Puede generar un automata de colados más grande y complejo en comparación con otros métodos de análisis sintáctico.

Y, por presentando conflictos desplazamiento-reducción y reducción-reducción.

4. Diseño y administración de una tabla de símbolos

Las tablas de símbolos (TS) son estructuras de datos que almacenan toda la información de símbolos o identificadores de un lenguaje, así como de sus atributos, los cuales se recogen y se guardan en la TS como el tipo de éstos, valor, dirección, dimensiones de los array, número y tipo de parámetro de los procedimientos, funciones y métodos a los que pertenecen, el tipo de acceso a los elementos de una clase (si es private, public, protected, etcétera). Es importante aclarar que estos atributos varían de lenguaje a lenguaje y también dependerá del elemento que se esté almacenando.

Los atributos de estos elementos se pueden resumir en los siguientes:

- **Tipo**: Ej. Entero, char, boolean, flotante, etc.
- **Valor**: Ej. 19, q, pingüino, Ø
- **Dirección** de memoria relativa en tiempo de ejecución

- Número de línea en el código en la que se encuentra
- Ámbito: Si pertenece a una función, método o procedimiento
- Número y tipo de parámetro en el ámbito
- Tipo de acceso a clase: Private, public, protected

A veces los atributos de un elemento se obtienen directamente del programa fuente, como en la sección de declaraciones de elementos; y otras veces se obtienen de forma implícita según el contexto en el que aparecen en el programa fuente.

En las TS, solamente se almacenan los identificadores, también conocidos como variables o no-terminales. Éstos son fundamentales en la mayoría de los lenguajes de programación y se utilizan para nombrar y acceder a los elementos en el código fuente. Por lo que, aunque se deduce, es importante mencionar que las TS no almacenan palabras reservadas o valores terminales como, por ejemplo, 'void', 'return', 'try' o 'catch' en Java. Estas palabras reservadas generalmente son guardadas en otra tabla y únicamente utilizadas por el analizador léxico.

~~Objetos~~ ~~funciones~~
~~Alejandro M. Gómez~~

Las TS, pese a ser estructuras de datos en sí, generalmente hacen uso auxiliar de otras estructuras de datos para lograr una búsqueda eficiente y una gestión adecuada de los símbolos o elementos.

Las operaciones que se utilizan para hacer uso de una TS son las de

- Inserción → Actualización
- Consulta/Búsqueda → Eliminación

La eficiencia de estas operaciones depende de la estructura de datos utilizada, y puede hacer que el compilador consuma mucho tiempo en los accesos a la misma.

Algunas de las estructuras de datos con las que se pueden implementar son listas lineales, árboles de búsquedas y las tablas de dispersión, también llamadas tablas hash, siendo éstas últimas la mejor opción pues con ella se pueden realizar operaciones de búsqueda, inserción y eliminación en tiempo real en la TS.

4.1 Acceso a la tabla de símbolos durante los distintas fases de un compilador

La importancia de las TS radica en su utilidad para las tres etapas de análisis de los compiladores e intérpretes, pues son introducidas para almacenar información durante estas etapas y, durante la generación de código intermedio, son utilizadas para obtener la información con la que se genera el código necesario. Además, también son usadas durante la fase de generación de código intermedio para identificar direcciones de elementos en la TS y en la fase de optimización de código también pueden ser accedidas, aunque no se ocupe realmente.

Las palabras reservadas (terminales) son habitualmente almacenadas en una tabla aparte y usadas exclusivamente en un analizador léxico. El analizador léxico separa el código en tokens, los cuales compara en primera instancia con las palabras reservadas. Si el token separado no es una palabra reservada, se asume que es un identificador, y, durante el análisis léxico sintáctico, este token se agrega a la TS.

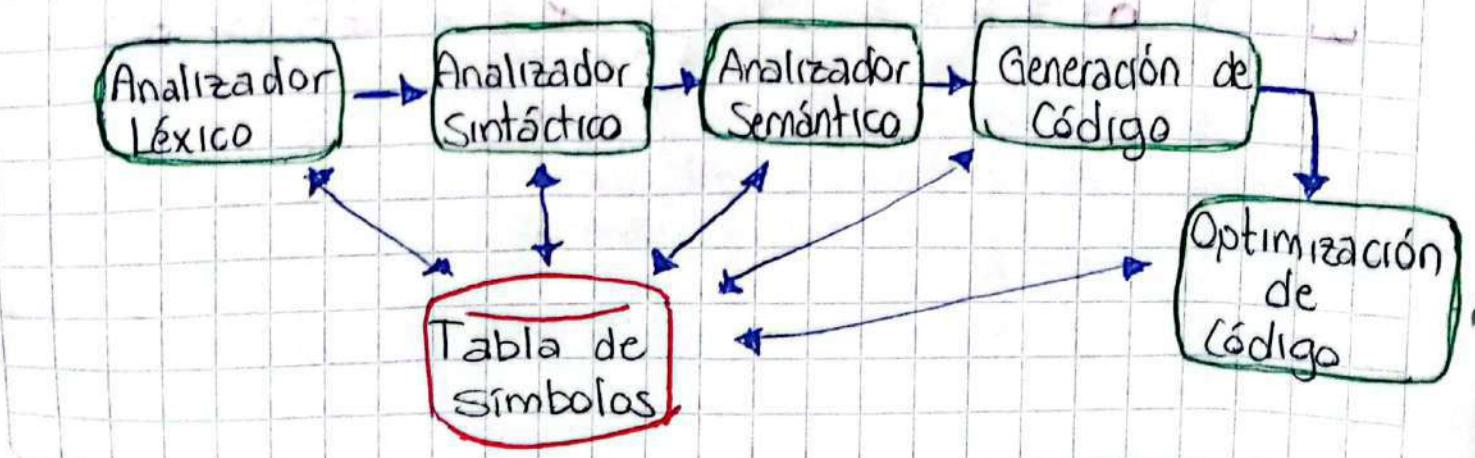


Figura 12. Acceso a las TS en las Fases del Compilador

Conforme van apareciendo nuevas declaraciones durante las etapas de análisis, los analizadores léxico y sintáctico insertarán nuevos registros en la TS, siempre evitando la repetición de registros ya existentes. Por su parte, el analizador semántico realiza comprobaciones sensibles al contexto gracias a la TS y en la etapa de generación de código intermedio se utilizan las direcciones asociadas a cada identificador en la TS para generar un programa equivalente al de entrada. En la etapa de optimización de código, no es necesario utilizar la TS, sin embargo, su uso es libre y nada impide que aún pueda ser accedida durante esa etapa.

4.2 Estructura de una tabla de símbolos

Como se mencionó con anterioridad, las TS hacen generalmente uso de una estructura de datos como una técnica para administrar mejor sus valores. Generalmente, una TS puede utilizar listas lineales, árboles de búsqueda binarios o tablas hash.

Sin embargo, para exemplificar una estructura elemental de un registro en una TS, se utilizará el siguiente formato:

< símbolo nombre, tipo, atributo >

Por ejemplo, se tiene la siguiente declaración de variable que almacena datos:

static float calificación
tipo de acceso tipo nombre

Entonces la TS guardará esta entrada en el siguiente formato:

<calificación, float, static>

4.3 Administración de una tabla de símbolos

Un compilador contiene dos tipos de tabla de símbolos. La tabla de símbolos global y las tablas de símbolos de alcance. La primera puede ser accedida por cualquier función o cualquier TS de alcance y las segundas se generan a partir de las operaciones o declaraciones internas que se hagan en cualquier función que pertenezca a la TS global.

A continuación, se mostrará un ejemplo extraído de la página JavaAtPoint, en el cual se muestra un código sencillo en el lenguaje C donde se declaran diferentes tipos de variables, como globales e independientes, internas dentro de funciones, etc.

```
1 int value = 10;  
2  
3 void sum_num()  
4 {  
5     int num_1;  
6     int num_2;  
7  
8     {  
9         int num_3;
```

11. int num_4;
12. }
13.
14. int num_5;
15.
16. {
17. int num_6;
18. int num_7;
19. }
20. }
21.
22. void sum_id()
23. {
24. int id_1;
25. int id_2;
26.
27. {
28. int id_3;
29. int id_4;
30. }
31.
32. int num_5;
33. }

Este código puede ser representado en una estructura jerárquica utilizando tablas de símbolos:

TS Global

nombre identificador tipo

value	var	int
sum-num	func	void
sum-id	func	void

TS sum-num

num_1	var	int
num_2	var	int
num_5	var	int

id-1	var	int
id-2	var	int
id-5	var	int

num_3	var	int
num_4	var	int

num_6	var	int
num_7	var	int

id-3	var	int
id-4	var	int

TS de
Alcance 1

TS de
Alcance 2

TS de
Alcance 3

Figura. 13.

Tablas de Símbolos Global y
Tablas de Símbolos de Alcance

La TS global contiene una variable global de tipo entero y dos funciones de tipo vacío, es decir, que no regresan ningún valor, sin embargo, dentro de estas funciones se efectúan declaraciones de más variables de tipo entero.

Estas variables de tipo entero declaradas dentro de las funciones no pueden pertenecer directamente en la TS global, pues no se encuentran declaradas en el código directamente de manera global. En su lugar, pertenecen a su respectiva TS de alcance.

5 Manejo de errores Sintácticos y su recuperación

Los compiladores serían mucho más sencillos de diseñar e implementar si este solo necesitara procesar programas correctos, mas este razonamiento escapa de la realidad pues los programadores no están exentos de cometer errores de múltiples indoles, relacionado a lo anterior se reconoce que un compilador es bueno si es capaz de ayudar al programador a identificar y localizar los errores que ah cometido, en algunos compiladores hasta se muestra un breve texto sugiriendo una corrección. Por ejemplo tomemos la estructura del lenguaje Java y realizemos el siguiente código:

```
1 package test;  
2 public class Test {  
3     public static void main (String [] args) {  
4         System.out.println ("Test"  
5     }  
6 }
```

Figura 14 Ejemplo impresion hello world java

En el código anterior el compilador podría devolver el siguiente mensaje: "Java.lang.RuntimeException: Código no compilable - el paquete System no existe en test.Test.main:4"

ABM

Se puede observar que se regresa una pequeña sinopsis sobre el error presentado Cnota: despues del primero encontrado se detiene el compilado y analisis pues en dicho existen otros dos errores), en este caso se indica en que linea se presento y por la frase "El paquete System no existe", el programador puede intuir o deducir que no escribió correctamente el nombre del paquete "System".

En el caso de Java, su compilador no sugiere que acciones se pueden realizar para la corrección de errores pero en algunos otros casos se puede imprimir un mensaje como el siguiente:

"Quizá quisiste decir System.out.println"; de la misma manera existen compiladores que su salida ante un error puede ser simplemente "Error en el archivo main, compilación interrumpida"; pero dicha salida no ayuda mucho al programador mas que darle a conocer que existe un error, este a la vez no estaría cumpliendo con los objetivos de los manejadores de errores, pero esto se abordara más adelante en este texto.

Los lenguajes de programación no se encargan de especificar como un compilador deberá actuar o responder ante errores, dicho comportamiento es responsabilidad del creador de dicho compilador.

Dichos errores pueden presentarse de diversas maneras y de diversos tipos como lo son:

- * **Lexicos:** El ejemplo anterior presenta uno de estos ya que fue mal escrita la palabra clave System.
- * **Sintáctico:** Se refiere por ejemplo, una expresión aritmética con parentesis no equilibrados (es decir faltan o sobran) por ejemplo en el ejemplo anterior nunca se cerro. Al igual que falte un punto y coma.
- * **Semantico:** Un operador aplicado a un operando incompatible.
- * **Lógico:** Una llamada infinitamente recursiva

En la mayoría de los casos gran parte de la detección y recuperación de errores esta centrado en la fase del análisis sintáctico debido a que gran parte de los errores son debidos a que los componentes léxicos no respetan las reglas gramaticales establecidas para el lenguaje de programación. Por lo anterior en esta sección se centrara exclusivamente en el manejo de errores sintácticos.

Antes de abordar el tema comenzaremos definiendo cuales objetivos deberá cumplir el manejador de errores que se encuentra dentro del analizador sintáctico que se puede resumir en lo siguiente:

- * Informar los errores captados de una manera clara y precisa y significativa.
- * Recuperarse de cada error presentado de una manera lo suficientemente rápida para detectar errores que se presenten posteriormente.
- * No deberá retrasar de forma significativa el procesamiento de programas correctos (es decir sin errores).

5.1 Información que deberá ofrecer un manejador de errores.

Lo mínimo que se deberá informar es el lugar exacto donde se encuentra dicho error en el programa fuente. Resulta muy común en los diferentes compiladores que se indique de la línea en donde radica el error, además de un apuntador a la posición del carácter donde inicia dicho error, en otras palabras el número de línea y los caracteres involucrados.

Como se mencionó anteriormente muchos compiladores cuentan con sugerencias para corregir el error encontrado, como por ejemplo mostrar un mensaje "Falta un punto y coma" o "Falta el parentesis derecho". Una vez detectado un error existen diferentes estrategias para recuperar errores, aunque cabe aclarar que ninguna es superior a los otros, sino que son situacionales y dependerá del desarrollador del compilador decidir cual se acopla mejor a su visión.

Una opción consiste en que al encontrar un ~~error~~, se cancele de golpe el análisis y se informe del error. Esto no es muy óptimo ya que una alternativa mejorada es que el compilador intente generar una entrada válida para continuar el procesamiento con la finalidad de intentar que esta sea correcta o manejada por el compilador. Aunque también puede ser contraproducente ya que el compilador con esta acción podría generar otros errores por si mismo, conocidos como errores falsos. Esto puede a la par generar errores semánticos falsos.

"Un ejemplo de lo anterior es el compilador PLTC"

Otra alternativa más conservadora consiste en ignorar en la ejecución los mensajes de error producidos de errores que estén demasiado cerca entre sí en la cadena de entrada. Una vez descubierto un error el compilador puede exigir que, obligatoriamente, se analicen o examinen varios componentes léxicos de forma correcta antes de permitir otro mensaje de error. Lo anterior es conveniente cuando existen demasiados errores en el código fuente, evitando analizar toda la entrada.

5.2 - Estrategias de recuperación de errores en analizadores sintácticos

Como se mencionó anteriormente, existen diversos métodos para la recuperación de errores y que ninguno es mejor que otros y tampoco se aceptan universalmente,

pero si figuran ciertos métodos de alta aplicabilidad como los que se abordaran a continuación:

5.2.1 Corrección global

Es considerado un método muy costoso respecto a su tiempo de ejecución y los requerimientos elevados de memoria, por lo que únicamente se plantea como teórico y no se emplea en la práctica.

Se fundamenta en que un compilador realice el número mínimo de cambios al analizar una cadena de entrada errónea. Es decir, que para una palabra X errónea, se busque otra palabra Y, usando árboles de análisis sintáctico, permitiendo que el número de inserciones/modificaciones y borrado sea mínimo. En la teoría sería lo más deseable y óptimo pero esto resulta en una carga demasiada elevada.

5.2.2 Modo Panico

En este al descubrirse un error, se procede a ignorar símbolos de entrada hasta descubrir alguno que pertenezca a un subgrupo designado especial, denominado componentes léxicos de sincronización (por ejemplo: "end", ";", ":", ")", "(", "E", "3", etc.) estos tienen muy bien delimitada su función en el programa fuente.

Su mayor beneficio consiste en la sencillez de su implementación y funcionara adecuadamente siempre y cuando el diseñador del compilador elija unos componentes de sincronización adecuados.

~~Alumno~~ ~~Objetivo~~

Ya que omite una gran cantidad de errores debido a que al encontrar un error comienza a ignorar símbolos. Siempre se debe considerar que dichos, a su vez pueden contener otros errores.

También otro de sus beneficios es garantizar la exclusión de los bucles infinitos.

5.2.3 Nivel de frase

En este método, al descubrirse un error, se intentará realizar una corrección local de la entrada restante (es decir, sustituir un punto por un punto y coma, eliminación de comas sobrantes, inserción de punto y coma faltante etc.) para permitir la continuación del análisis sintáctico.

El mayor inconveniente de este radica en que es complicada la elección de la sustitución que se empleara para los casos de errores ya que dicha acción puede llevar a bucles infinitos.

5.2.4 Producciones de error

Es utilizado cuando se tiene previamente conocimiento de cuales pudiesen ser los errores mas frecuentes que se pudiesen presentar en el código fuente. Consiste en aumentar la gramática del lenguaje con producciones que generen las construcciones erroneas. Esto tendrá de resultado una nueva GLC (gramática libre de contexto) con las producciones de error para la construcción del analizador sintáctico. Esto permite la generación de mensajes de error mas adecuadas y recuperaciones mas eficientes.

6 Generadores de código para analizadores sintácticos: Yacc, Bison

En el contexto del desarrollo de compiladores y en concreto en la fase de análisis sintáctico de lenguajes de programación, los generadores de código tienen un papel muy importante al momento de automatizar la creación de analizadores sintácticos, permitiendo al desarrollador crear dicho sin demasiada complicación a la par de reducir el tiempo de desarrollo, su costo así como también reducir los errores. Las dos herramientas que destacan en este rubro son YACC y BISON, ambos son generadores de analizadores sintácticos que toman como entrada una gramática libre de contexto y producen como salida un analizador sintáctico en C. A continuación se muestra la interacción de YACC/BISON con los demás procesos de análisis.

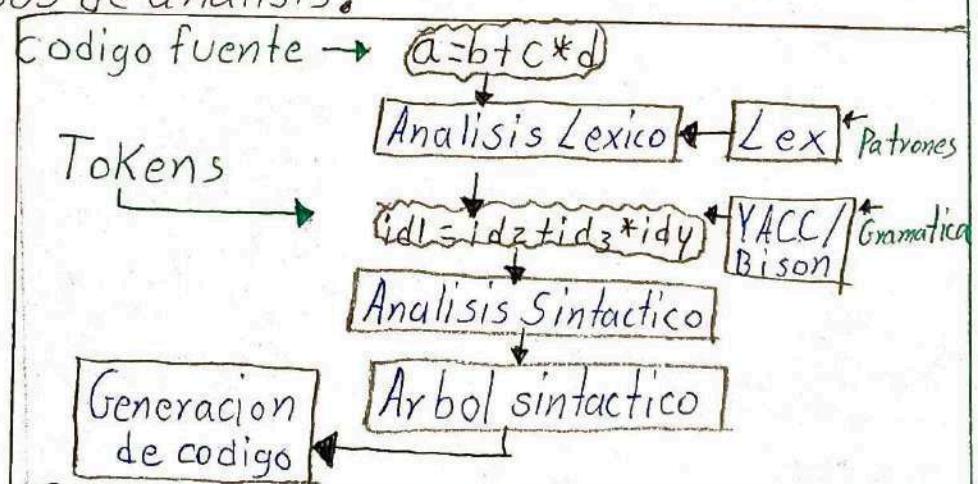


Figura 15 Interacción Yacc/bison con procesos de análisis

Ahora a continuación, se hablará sobre ambos.

6.1 Yacc

Su nombre es un acrónimo de "Yet another Compiler Compiler", que traducido vendría siendo "Otro compilador más de compiladores" siendo uno de los pioneros en dicha rama. Desarrollado originalmente en los laboratorios Bell de AT&T y lanzado en la década de 1970. Su función como su nombre dice, es generar analizadores sintáticos en el lenguaje de programación C (mismo que YACC) a partir de una especificación de una gramática formal (concretamente una gramática libre de contexto).

Ya que, a partir de la GLC, YACC puede generar un analizador sintático que pueda reconocer y descomponer el código fuente en unidades sintáticas significativas.

YACC está diseñado para analizar un código escrito en C y generar un analizador sintático igualmente escrito en C. Dicho analizador está hecho para funcionar complementariamente con un analizador léxico generado típicamente por LEX y depende de ciertas características compartidas con éste (tipos de tokens, la variable yyval, etcétera). Para ello, YACC llama a la rutina yylex para trabajar a la par con un analizador léxico.

Generalmente los archivos de YACC tienen una extensión ".y".

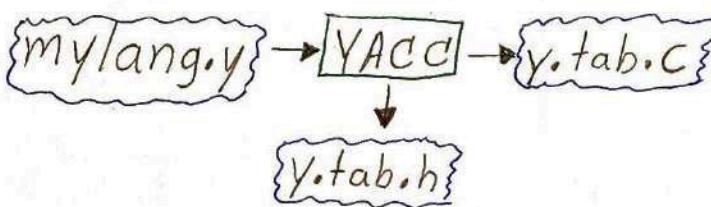


Figura 16 Ejemplo archivo YACC

El archivo de extensión '.y' contiene las especificaciones de una gramática, especificaciones que el desarrollador define. Después se llama a YACC sobre este archivo '.y' para generar los archivos llamados 'y.tab.h' y 'y.tab.c', los cuales contienen miles de líneas de código generado en C que crean a un analizador sintáctico LALR(1) para la gramática definida, incluido en el código para las acciones especificadas. Para generar el analizador sintáctico, tan solo se debe compilar el archivo '.c' y enlazarlo con el resto del código C. Esto último se hace con el comando de C llamado 'gcc' en las distribuciones GNU/Linux).

6.2 Bison

Es una implementación de software libre escrita por la Free Software Foundation, este mismo se volvió muy popular para el desarrollo de compiladores. Bison es compatible con YACC pero el primero ofrece algunas mejoras y características adicionales. Como por ejemplo:

* **Manejo de conflictos gramaticales:** Bison proporciona mecanismos para resolver conflictos gramaticales lo que permite el lidiar con ambigüedades en las gramáticas de manera más eficaz.

* **Mayor personalización y extensibilidad:** Permite a los desarrolladores el personalizar y extender la funcionalidad de los analizadores que genera, volviéndolo adecuado para casos que requieran características específicas o modificaciones en el proceso de análisis sintáctico.

Entonces dicho lo anterior se puede venir en mente la siguiente pregunta ¿Cuál se debe elegir o cuales mejor?

Generalmente no hay una respuesta clara ya que a grandes rasgos dependerá de las necesidades del proyecto y las preferencias del desarrollador. Ambas son igualmente herramientas valiosas para el desarrollo de compiladores y sistemas que involucran el análisis sintáctico de lenguajes de programación. Además es relevante mencionar que existen alternativas para otros lenguajes de programación de salida como lo es ANTLR para Java, PLY para Python, entre otros. Como último se menciona que tanto Yacc como Bison tienen de salida un analizador sintáctico ascendente.

Conclusiones

En esta monografía, se abordaron los temas de precedencia de operadores, analizadores sintácticos ascendentes y descendentes, tablas de símbolos, errores sintácticos y los generadores de analizadores sintácticos YACC y Bison; gracias a su exhaustiva investigación y profunda comprensión, se ha proporcionado una visión integral de las técnicas y herramientas esenciales en la construcción y análisis de lenguajes formales en el equipo.

Los conceptos presentados en la monografía nos han permitido al equipo explorar en profundidad aspectos fundamentales de la gramática formal y la generación de analizadores sintácticos.

Todos estos temas han ampliado el conocimiento general y científico en el campo de la teoría de compiladores y la construcción de lenguajes de programación.

Además, se realizaron ejercicios prácticos relacionados con analizadores sintácticos ascendentes y descendentes así como con YACC y Bison, que nos han permitido aplicar estos conocimientos en situaciones reales como el desarrollo de calculadoras o un reconocedor de oraciones simples en inglés. Estos ejercicios han reforzado la comprensión de los conceptos teóricos y han fomentado la adquisición de habilidades prácticas en la creación y análisis de gramáticas, lo que es esencial en el desarrollo de lenguajes de programación y sistemas de compilación.

Los conocimientos adquiridos en estos temas son esenciales para aquellos que se dedican al desarrollo de compiladores, intérpretes y lenguajes de programación, y son un recurso valioso en el campo de la informática y la ciencia de la computación.

Bibliografía

Programación en Java/Precedencia de operadores - Wikilibros. (s. f.).

https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Java/Precedencia_de_operadores

Área de Computación ISCO Yoro-Yoro. (2014, 30 julio). *Precedencia de operadores*

[Vídeo]. YouTube. <https://www.youtube.com/watch?v=0mH04VRWqkc>

Llamas, L. (2023). Precedencia de operadores. *Luis Llamas*.

<https://www.luisllamas.es/programacion-precedencia-operadores/>

Asociatividad de Operadores _ AcademiaLab. (s. f.). <https://academialab.com/enciclopedia/asociatividad-de-operadores/>

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

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

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

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

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

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

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

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers, principles, techniques, and tools* (2^a ed.). Pearson Education.
<https://drive.google.com/file/d/0B1MogsyNAsj9elVzQWR5NWVTSVE/view?resourcekey=0-zoBDMpzTafr6toxDuQLNUg> (Obra original publicada en 1986)

Analizador, U., & St, A. (n.d.). *Analizador Sintáctico Ascendente*.

<https://dlsiis.fi.upm.es/procesadores/Documentos/AStLR.pdf>

(S/f). Informatica.uv.es. Recuperado el 9 de octubre de 2023, de

<http://informatica.uv.es/docencia/iigua/asignatu/2000/PL/2007/tema5.pdf>

(S/f-b). Uhu.es. Recuperado el 9 de octubre de 2023, de

http://www.uhu.es/francisco.moreno/gii_pl/docs/Tema_4.pdf

Sierra, A., & Rojas, S. G. (s/f). *Traductores, Compiladores e Intérpretes 1 Análisis*

Sintáctico Realizados por: María del Mar. Uma.es. Recuperado el 9 de octubre de 2023, de <http://www.lcc.uma.es/~galvez/ftp/tci/tictema3.pdf>

Nº, C., Oviedo, A., Manuel, J., & Lovelle, C. (s/f). *Análisis Sintáctico en Procesadores de Lenguaje*. Uniovi.es. Recuperado el 9 de octubre de 2023, de

<http://di002.edv.uniovi.es/~cueva/publicaciones/monografias/Cuaderno-61-Matematicas-Sintactico.pdf>

Wayne Cochran. (2014, 26 enero). Introduction to YACC [Vídeo]. YouTube.

<https://www.youtube.com/watch?v=yTXCPGAD3SQ>

GeeksforGeeks. (2023). Introduction to YACC. GeeksforGeeks.

<https://www.geeksforgeeks.org/introduction-to-yacc/>

3.1.2 *Estrategias de recuperación de errores*. (s. f.).

http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/312_estrategias_de_recuperacion_de_errores.html

3.1.1 Manejo de los errores sintácticos. (s. f.).

http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/311_manejo_de_los_errores_sintcticos.html

Simmross, F. S. W. (s. f.). El generador de analizadores sintácticos yacc. *Departamento de Informática de la Universidad de Valladolid.*

<https://www.infor.uva.es/~mluisa/talf/docs/lab0/L8.pdf>

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

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

[Handwritten signatures]

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

PRACTICA NO.	NOMBRE DE LA PRÁCTICA	FECHA DE ENTREGA
1	Analizadores sintácticos ascendentes y descendentes	10/10/2023

1**INTRODUCCION**

Con el desarrollo de esta práctica se busca comprender de una mejor manera a los analizadores sintácticos, específicamente los analizadores sintácticos descendentes y ascendentes. Para cumplir con este propósito, se desarrollarán ejercicios prácticos en los cuales se construirán estos analizadores; se utilizarán herramientas como ANTLR, FLEX y Bison. Con estos ejercicios se comprenderá como funcionan estos analizadores y como se aplican para la validación de estructuras sintáctica en lenguajes formales.

2**OBJETIVO**

El objetivo de esta práctica es demostrar, mediante ejercicios prácticos, el funcionamiento de un analizador sintáctico ascendente y descendente, así como comprender cuáles son los objetivos y las funciones de un analizador sintáctico.

3**FUNDAMENTO**

El desarrollo de un analizador sintáctico descendente LL es crucial para comprender cómo las máquinas pueden analizar y validar la estructura sintáctica de un lenguaje de programación o un conjunto de reglas gramaticales, este tipo de analizador sintáctico comienza desde el símbolo inicial de una gramática y se desplaza hacia abajo en la jerarquía de las reglas de producción para reconocer la estructura sintáctica de una cadena de entrada. A su vez lo anterior utiliza gramáticas formales, especialmente las gramáticas libres de contexto, para describir las reglas sintácticas de los lenguajes de programación. Estas gramáticas son la base para el diseño del analizador sintáctico descendente LL, ya que definen cómo se deben construir las cadenas válidas en el lenguaje. Las gramáticas emplean las reglas de producción establecen cómo se pueden formar las cadenas válidas dentro del lenguaje. Estas reglas son esenciales para el desarrollo del analizador sintáctico y determinan la estructura sintáctica permitida.

Para implementar un analizador sintáctico descendente LL, creamos una tabla de análisis predictivo que relaciona los símbolos de entrada con las reglas de producción a aplicar. Esta tabla es crucial para avanzar en el análisis sintáctico y decidir qué regla de producción usar en cada paso.

El análisis sintáctico ascendente es un enfoque utilizado para determinar si una cadena de texto, denotada como "x," pertenece al lenguaje definido por una gramática, comúnmente llamada "G."

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

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

Para llevar a cabo este análisis, se siguen ciertos criterios específicos:

- Se comienza dividiendo la cadena "x" en sus elementos terminales, que son las partes fundamentales que constituyen la frase.
- Luego, se seleccionan de manera estratégica las reglas gramaticales de la gramática "G" que se aplicarán a la cadena.
- La aplicación de estas reglas se realiza en el orden inverso, es decir, se realizan derivaciones partiendo desde las partes más a la derecha de la cadena, siguiendo el principio de "Right Most Derivation."
- El procesamiento de la cadena se lleva a cabo de izquierda a derecha, asegurándose de que las reglas gramaticales se apliquen de manera coherente.
- El objetivo final del análisis es tratar de llegar al "axioma," que es la regla gramatical inicial de la gramática "G." Si se logra alcanzar el axioma, se confirma que la cadena "x" pertenece al lenguaje definido por la gramática y, por lo tanto, se construye el árbol de análisis sintáctico correspondiente. Sin embargo, si no se logra llegar al axioma de manera válida, se identifica un error en la estructura de la cadena.

Los analizadores sintácticos ascendentes tienen la capacidad de determinar de manera inteligente cuál regla de producción se debe aplicar en cada paso, basándose en los elementos terminales que se encuentran al principio de la cadena de entrada que se está analizando. Este enfoque permite realizar el análisis con una complejidad lineal, lo que significa que el tiempo necesario para analizar una cadena de entrada es proporcional al tamaño de la cadena ($O(n)$).

Estos analizadores se conocen comúnmente como analizadores LR (K) o analizadores de reducción-desplazamiento. En esencia, pueden "reducir" la cadena a su forma más simple y, al mismo tiempo, "desplazar" elementos para avanzar en el proceso de análisis. Esta eficiencia en el análisis sintáctico los hace ampliamente utilizados en la construcción de compiladores y analizadores de lenguaje.

4	MATERIALES NECESARIOS
	<ul style="list-style-type: none">• Computadora• Máquina virtual con la distribución de Linux de su preferencia.• Editor de texto en la distribución de Linux• Editor de código (opcional)• ANTLR• Flex• Bison

5	DESARROLLO
	<p>ANALIZADOR SINTÁCTICO DESCENDENTE</p> <p>ALGORITMO</p> <ol style="list-style-type: none">1. Inicializar el analizador léxico y sintáctico de ANTLR para la gramática "Expr".2. Crear una función principal que representará el punto de entrada para el análisis sintáctico del

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

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

[Handwritten signatures]

programa.

3. En la función principal, iniciar el análisis sintáctico llamando al inicio de la regla "prog".
4. La regla "prog" permite una secuencia de expresiones separadas por saltos de línea.
5. En cada iteración del bucle, llame a la regla "expr" para analizar una expresión.
6. Dentro de la regla "expr", implemente las reglas de producción definidas en la gramática:
 - Si la expresión contiene una multiplicación o división, analice las subexpresiones a ambos lados del operador '*' o '/'.
 - Si la expresión contiene una suma o resta, analice las subexpresiones a ambos lados del operador '+' o '-'.
 - Si la expresión es un número entero (INT), guárdealo como un nodo en el árbol de sintaxis abstracta (AST).
 - Si la expresión está encerrada entre paréntesis, analice la subexpresión dentro de los paréntesis recursivamente.
7. Cuando se complete el análisis sintáctico de una expresión, continúe analizando la siguiente expresión en la secuencia si existen más.
8. Se repite el proceso hasta que todas las expresiones en el programa hayan sido analizadas.
9. Si se encuentra un error sintáctico, se maneja la excepción correspondiente y se muestra un mensaje de error informativo.
10. Al finalizar el análisis sintáctico del programa y, si no se han encontrado errores, tendrá un AST que representa la estructura sintáctica del programa.

PSEUDOCÓDIGO

```
Algoritmo Analizador_Descendente
    Definir expresion Como Cadena;
    Definir i Como Entero;
    Definir parentesis Como Entero;
    Definir caracter Como Caracter;
    Escribir Sin Saltar "Ingrese una expresión: ";
    Leer expresion;

    i <- 1;
    parentesis <- 0;

    Mientras i <= Longitud(expresion) Hacer
        caracter <- Subcadena(expresion, i, 1);
        Si caracter = '(' Entonces
            parentesis <- parentesis + 1;
        SiNo
            Si caracter = ')' Entonces
                parentesis <- parentesis - 1;
            SiNo
                Si parentesis < 0 Entonces
                    Escribir "Error: Paréntesis no
balanceados.";
```



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
Fin Si
Fin Si
    Si caracter <> '+' Y caracter <> '-' Y caracter <>
'*' Y caracter <> '/' Y No EsDigito(caracter) Entonces
        Escribir "Error: Caracter no permitido en
la expresión.";
        i= Longitud(expresion);
    Fin Si
Fin Si

    i <- i + 1;

Fin Mientras

Si parentesis <> 0 Entonces
    Escribir "Error: Paréntesis no balanceados.";
Sino
    Escribir "La expresión es válida.";
FinSi

FinAlgoritmo

SubProceso VerificarDigito <- EsDigito(c)
    Definir x Como Logico;
    x <- Falso;
    Si c >= '0' Y c <= '9' Entonces
        x <- Verdadero;
    FinSi
    VerificarDigito <- x; // Cambia la asignación a un retorno
FinSubProceso
```

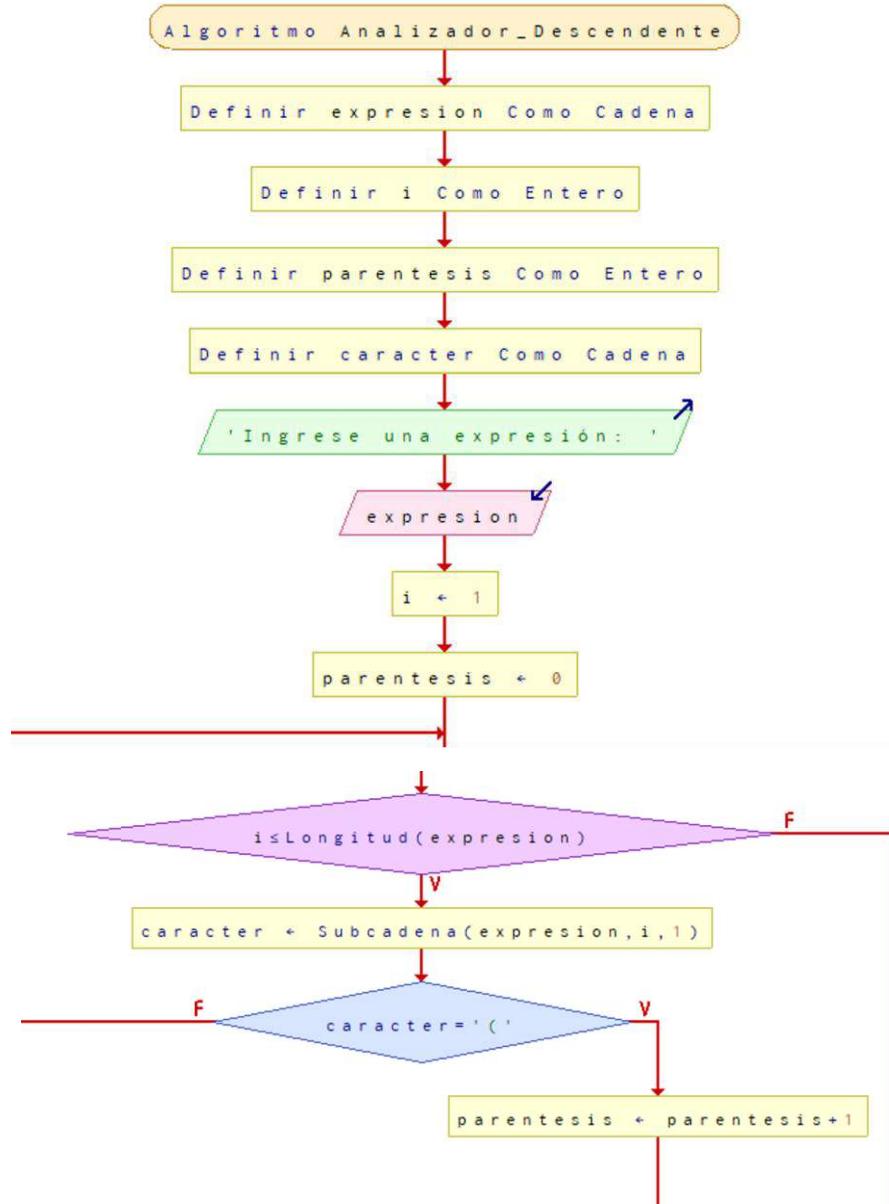
PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II

AUTORES:

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

[Handwritten signatures]

DIAGRAMA DE FLUJO





TECNOLÓGICO
NACIONAL DE MÉXICO

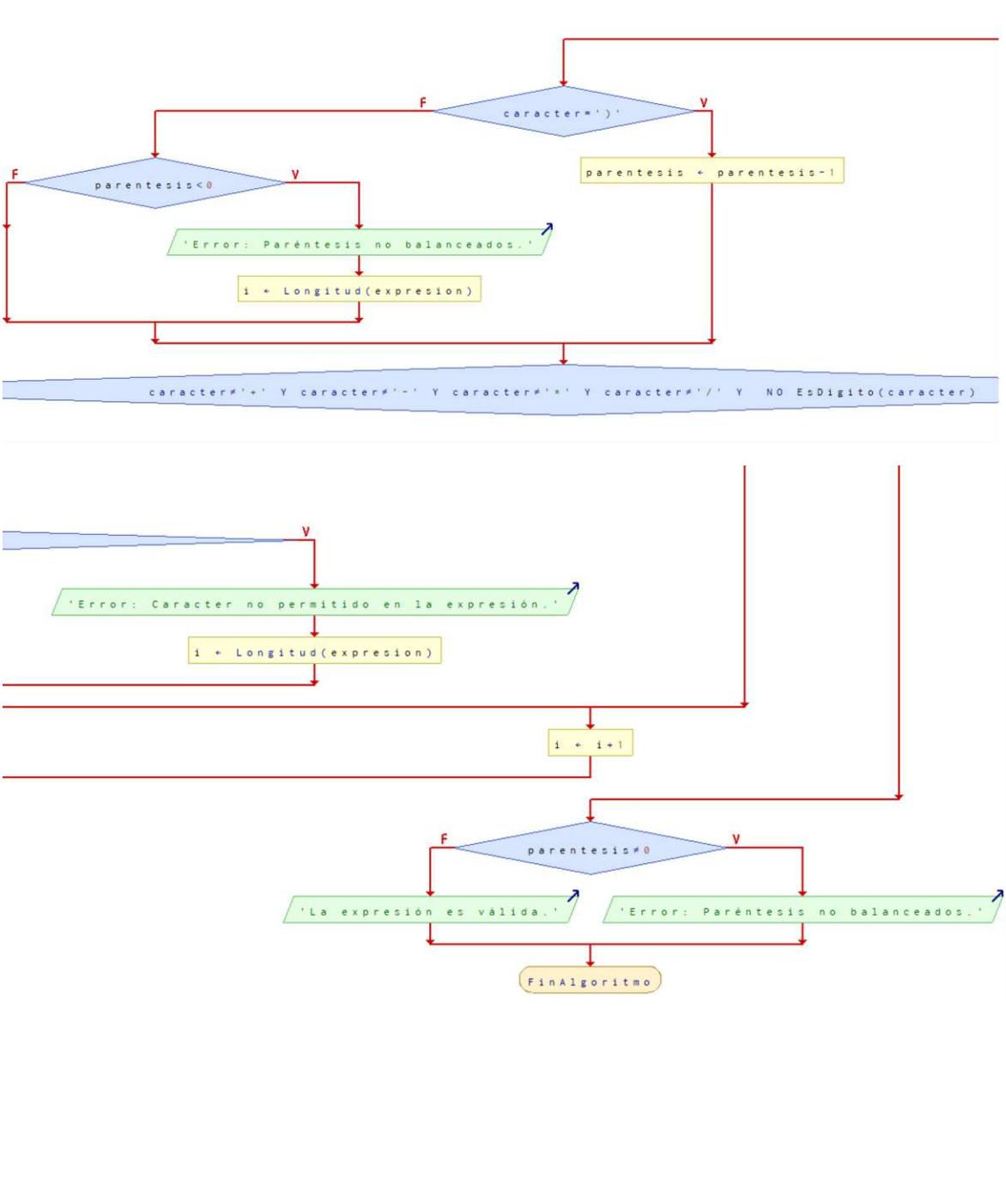
INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

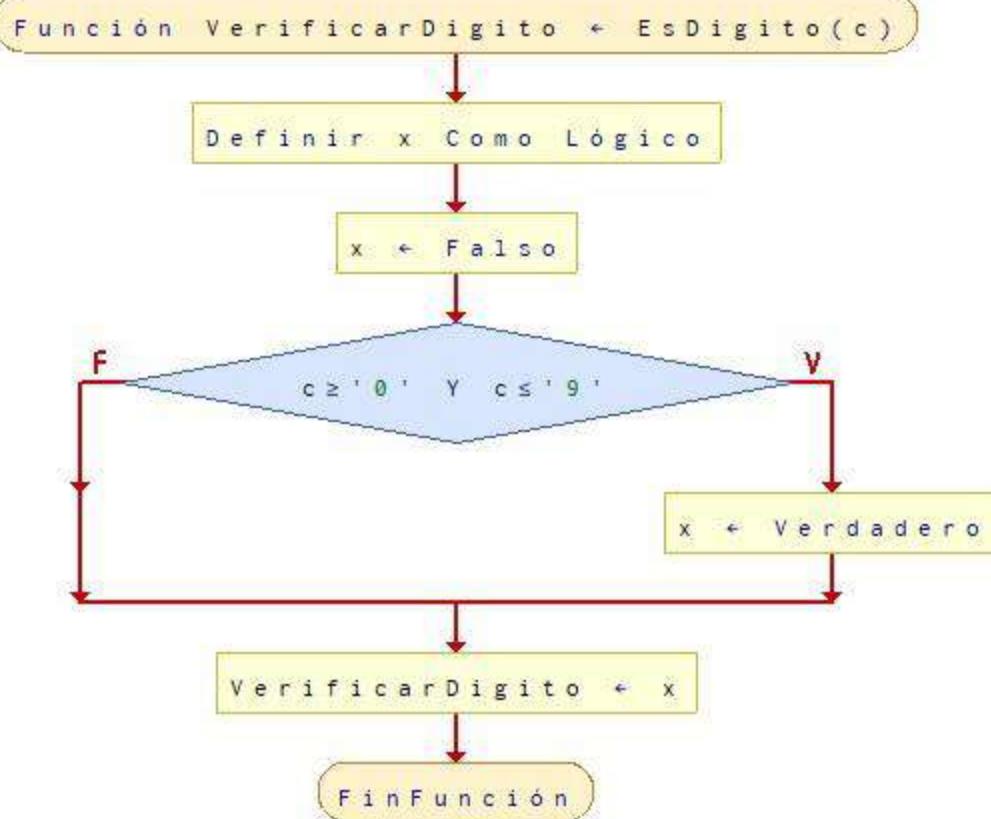
Alma Gabriela Ponce Morales
Anthony Gómez Cabañas
Cristhian Alberto Ortega Hernández
Miguel Ángel Ruiz López



PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II

AUTORES:

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



Para comprender cómo funcionan los analizadores sintácticos descendentes, es necesario examinar la ejecución de un ejercicio específico. En esta ocasión, se optó por no utilizar YACC, ya que es conocido por generar analizadores sintácticos ascendentes, como los del tipo LALR (Look-Ahead Left-to-Right) y SLR (Simple LR). Estos analizadores adoptan un enfoque bottom-up para construir el árbol de análisis sintáctico a partir de la entrada.

En su lugar, se eligió ANTLR como herramienta, ya que es ampliamente reconocida y utilizada para crear analizadores sintácticos descendentes LL. ANTLR permite definir gramáticas de lenguaje en un formato simple y genera código en varios lenguajes de programación.

Esta elección se consideró óptima debido al objetivo del ejercicio, que busca comprender cómo se descompone una entrada según la gramática y cómo se representa esta estructura jerárquica en forma de árbol de análisis sintáctico (AST). El AST resultante es extremadamente útil para visualizar la sintaxis del lenguaje, permitiendo la inspección de cómo los tokens se organizan en una estructura de árbol.

La gramática, en este contexto, establece las reglas para estructurar las sentencias en un lenguaje



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

de programación. En este caso, la gramática "Expr" describe cómo se deben formar las expresiones aritméticas, incluyendo operaciones básicas como suma, resta, multiplicación y división, así como el uso de paréntesis. A continuación, se presenta el código implementado en el archivo correspondiente.

```
grammar Expr;
prog: (expr NEWLINE)* ;
expr: expr ('*' | '/') expr
    | expr ('+' | '-') expr
    | INT
    | '(' expr ')'
    ;
NEWLINE : [\r\n]+ ;
INT : [0-9]+ ;
```

Observado lo anterior se puede empezar con la explicación de la gramática, se puede observar la regla “prog” que establece que va a estar formado por una serie de expresiones separados por saltos de línea.

- La regla "expr" se utiliza para definir expresiones aritméticas. Las que se encuentran definidas son: Una expresión seguida de un operador de multiplicación ('*') o división ('/'), seguida de otra expresión.
- Una expresión seguida de un operador de suma ('+') o resta ('-'), seguida de otra expresión.
- Un número entero (INT).
- Una expresión encerrada entre paréntesis '(' y ')'.

Para lograr esto, es necesario establecer un conjunto de métodos en donde se realizarán el proceso correspondiente a la opción seleccionada, entonces se siguió los siguientes pasos. Finalizando de explorar este código se puede encontrar el término "NEWLINE", se utiliza para representar saltos de línea en el programa, y "INT" se utiliza para representar números enteros. De manera posterior al utilizar el comando.

antlr Expr.g4

Se generan varios archivos, entre los cuales uno de los más interesantes y digno de análisis es el archivo del parser que tiene un nombre similar al establecido en la gramática, como ExprParser.java si el código generado es en java.

Dentro del archivo del parser, se encuentran las reglas gramaticales que se definieron en la gramática original. ANTLR se encarga de generar automáticamente el código necesario para llevar a cabo el análisis sintáctico basado en esas reglas. Además, tienes la capacidad de personalizar este archivo del parser para incorporar acciones semánticas específicas cuando se detecten las reglas gramaticales en la entrada. Donde se va a proceder a explicar un poco de este código.

// Generated from Expr.g4 by ANTLR 4.13.1



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
import org.antlr.v4.runtime.Lexer;
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.atn.*;
import org.antlr.v4.runtime.dfa.DFA;
import org.antlr.v4.runtime.misc.*;

@SuppressWarnings({"all", "warnings", "unchecked", "unused", "cast",
"CheckReturnValue", "this-escape"})
public class ExprLexer extends Lexer {
    static { RuntimeMetaData.checkVersion("4.13.1",
RuntimeMetaData.VERSION); }

    protected static final DFA[] _decisionToDFA;
    protected static final PredictionContextCache _sharedContextCache
=
        new PredictionContextCache();
    public static final int
        T__0=1, T__1=2, T__2=3, T__3=4, T__4=5, T__5=6, NEWLINE=7,
INT=8;
    public static String[] channelNames = {
        "DEFAULT_TOKEN_CHANNEL", "HIDDEN"
    };

    public static String[] modeNames = {
        "DEFAULT_MODE"
    };

    private static String[] makeRuleNames() {
        return new String[] {
            "T__0", "T__1", "T__2", "T__3", "T__4", "T__5",
"NEWLINE", "INT"
        };
    }
    public static final String[] ruleNames = makeRuleNames();

    private static String[] makeLiteralNames() {
        return new String[] {
            null, "'*'", "'/'", "'+'", "'-'", "'('", "')'"
        };
    }
    private static final String[] _LITERAL_NAMES = makeLiteralNames();
    private static String[] makeSymbolicNames() {
        return new String[] {
            null, null, null, null, null, null, null,
```

**PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II****AUTORES:**

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

```
"NEWLINE", "INT"
    };
}
private static final String[] _SYMBOLIC_NAMES =
makeSymbolicNames();
public static final Vocabulary VOCABULARY = new
VocabularyImpl(_LITERAL_NAMES, _SYMBOLIC_NAMES);
```

En primer lugar, se importan las bibliotecas necesarias para utilizar ANTLR y trabajar con su runtime. Estas importaciones incluyen clases como Lexer, CharStream, Token, TokenStream, y otras relacionadas con la infraestructura interna de ANTLR, como el modelo de red de transiciones ATN y autómatas finitos deterministas (DFA).

Se define la clase ExprLexer, que representa el analizador léxico generado. Dentro de la clase ExprLexer, hay un bloque estático que verifica la versión específica de ANTLR (4.13.1) para garantizar la compatibilidad.

Se definen constantes y estructuras de datos que se utilizarán en el análisis léxico, incluyendo nombres de tokens, canales de tokens y modos de análisis. Además, se definen métodos que describen las reglas gramaticales y literales utilizadas para reconocer tokens en la entrada. Estas reglas se basan en la gramática Expr.g4 y se utilizan para generar tokens correspondientes a los símbolos gramaticales. De manera posterior se puede topar con el siguiente fragmento de código:

```
/**
 * @deprecated Use {@link #VOCABULARY} instead.
 */
@Deprecated
public static final String[] tokenNames;
static {
    tokenNames = new String[_SYMBOLIC_NAMES.length];
    for (int i = 0; i < tokenNames.length; i++) {
        tokenNames[i] = VOCABULARY.getLiteralName(i);
        if (tokenNames[i] == null) {
            tokenNames[i] =
VOCABULARY.getSymbolicName(i);
        }
        if (tokenNames[i] == null) {
            tokenNames[i] = "<INVALID>";
        }
    }
}

@Override
@Deprecated
public String[] getTokenNames() {
```



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
        return tokenNames;
    }

    @Override

    public Vocabulary getVocabulary() {
        return VOCABULARY;
    }

    public ExprLexer(CharStream input) {
        super(input);
        _interp = new
LexerATNSimulator(this,_ATN,_decisionToDFA,_sharedContextCache);
    }

    @Override
    public String getGrammarFileName() { return "Expr.g4"; }

    @Override
    public String[] getRuleNames() { return ruleNames; }

    @Override
    public String getSerializedATN() { return _serializedATN; }

    @Override
    public String[] getChannelNames() { return channelNames; }

    @Override
    public String[] getModeNames() { return modeNames; }

    @Override
    public ATN getATN() { return _ATN; }

    public static final String _serializedATN =
"\u0004\u0000\b'\u0006\xffff\xffff\u0002\u0000\u0007\u0000\u0002\u0001"+
"\u0007\u0001\u0002\u0002\u0007\u0002\u0002\u0003\u0007\u0003\u0000\u0004"+
"\u0007\u0004\u0002\u0005\u0007\u0005\u0002\u0006\u0007\u0006\u0000\u0007"+
"\u0007\u0001\u0000\u0001\u0000\u0001\u0000\u0001\u0001\u0001\u0001\u0000\u0002"+
"\u0007\u0007\u0001\u0000\u0001\u0000\u0001\u0000\u0001\u0001\u0001\u0001\u0000\u0002"+
```



TECNOLÓGICO
NACIONAL DE MÉXICO®

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

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

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

```
"\u0000\u0000\u0000\u0000\u00019\u001a\u0005 (\u0000\u0000\u001a\n\u0001\u0000"+
    "\u0000\u0000\u001b\u001c\u0005 )\u0000\u0000\u001c\f\u0001\u0000\u0000"+
    "\u0000\u001d\u001f\u0007\u0000\u0000\u000e\u001d\u0001\u0000\u0000"+
        "\u0001\u000f \u0001\u0000\u0000\u0000\u0000\u0000"+
    "\u0001\u0001\u0001\u0000\u0000\u0000 "+
    "\u0001\u0000\u0000\u0000\u0000!\u000e\u0001\u0000\u0000\u0000\u0000\"$\u0007
\u0001"+
    "\u0000\u0000#\\"\u0001\u0000\u0000\u0000$%\u0001\u0000\u0000\u0000%#\u0001"+
    "\u0000\u0000\u0000%&\u0001\u0000\u0000\u0000&\u0010\u0001\u0000\u0000\u0000"+
        "\u0000\u0003\u0000 %\u0000";
public static final ATN _ATN =
    new
ATNDeserializer().deserialize(_serializedATN.toCharArray());
static {
    _decisionToDFA = new DFA[_ATN.getNumberOfDecisions()];
    for (int i = 0; i < _ATN.getNumberOfDecisions(); i++) {
        _decisionToDFA[i] = new
DFA(_ATN.getDecisionState(i), i);
    }
}
```

Primero, hay un bloque de comentarios que menciona que el uso de tokenNames está desaprobado, y en su lugar, se debe usar VOCABULARY. Esto se refiere a una forma anterior de definir los nombres de los tokens en ANTLR. Los nombres de los tokens son importantes para identificar los elementos reconocidos en el análisis léxico. Sin embargo, ahora se recomienda utilizar VOCABULARY en lugar de tokenNames para acceder a esta información.

Luego, se define una matriz llamada tokenNames que se utiliza para almacenar los nombres de los tokens. Estos nombres se asignan a partir de información en VOCABULARY. Esto es necesario para proporcionar nombres más descriptivos a los tokens en lugar de los simples valores numéricos que se utilizan internamente.

Además, se anula el método getTokenNames() para devolver los nombres de los tokens almacenados en tokenNames. Esto asegura que los nombres de los tokens sean accesibles en el código generado.

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

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

[Handwritten signatures]

Por último, se sobrescribe el método `getVocabulary()` para devolver el vocabulario definido en `VOCABULARY`, que es esencial para mapear los nombres simbólicos de los tokens a los valores numéricos correspondientes.

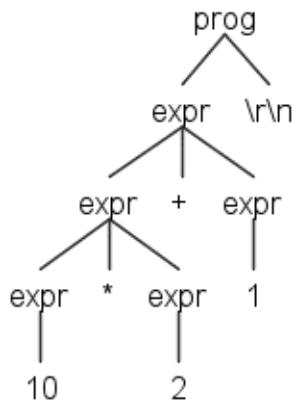
Explicado esto se puede proceder a las pruebas unitarias para poner a prueba lo generado anteriormente, en este documento se encuentra lo siguiente:

```
10*2+1
```

Se ejecuta entonces tanto el archivo `.g4` junto con el archivo de las pruebas unitarias con el siguiente comando:

```
grun Expr_prog -gui < example.txt
```

Lo anterior visualizar el árbol de análisis sintáctico resultante en una interfaz gráfica de usuario (GUI). Esto puede ser útil para depurar y comprender cómo se está analizando sintácticamente el código fuente según la gramática definida. Generando el siguiente árbol.



En este caso, se tiene la expresión `10*2+1` que se quiere analizar siguiendo las reglas definidas en la gramática.

La regla "`expr`" es la que descompone la expresión matemática en sus componentes. Dentro de "`expr`", hay varias opciones, incluyendo la multiplicación y la división, representadas por `10*2`, y la suma, representada por `+1`.

El proceso de análisis sintáctico sigue estas pautas. Primero, la computadora reconoce `10` como un número entero (INT), luego identifica `*` como el operador de multiplicación y finalmente reconoce `2` como otro número entero. Por lo tanto, se entiende que aquí se realiza la multiplicación `10*2`.

Luego, se analiza si hay una suma o resta en la expresión, que en este caso es `+1`. La computadora reconoce `+` como el operador de suma y `1` como otro número entero (INT). Por lo tanto, se comprende que aquí se realiza la suma `+1`.

Finalmente, la computadora combina estos componentes y construye un árbol de análisis sintáctico que refleja el orden de las operaciones. Primero se realiza la multiplicación `10*2` y luego se suma `+1` al resultado. De esta manera, la computadora comprende cómo resolver la expresión de acuerdo



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

con la gramática definida.

ANALIZADOR SINTÁCTICO ASCENDENTE

Algoritmo

Proceso Principal

1. Mostrar "Iniciando análisis..."
2. Llamar al subproceso Analizador_Lexico:
 - a. Definir los tokens que se esperan en la entrada.
 - b. Leer una entrada de texto proporcionada por el usuario.
 - c. Iniciar un bucle mientras haya entrada disponible:
 - Si la entrada está en la lista de tokens esperados:
 - Mostrar "Token reconocido: " seguido de la entrada.
 - De lo contrario:
 - Mostrar "Error: Carácter desconocido".
 - Leer la siguiente entrada.
 - d. Finalizar el bucle cuando no haya más entrada.
3. Llamar al subproceso Analizador_Sintactico:
 - a. Definir la gramática que se espera en la entrada.
 - b. Leer una entrada de texto proporcionada por el usuario.
 - c. Iniciar un bucle mientras haya entrada disponible:
 - Si la entrada cumple con la gramática especificada:
 - Mostrar "Entrada sintácticamente correcta: " seguido de la entrada.
 - De lo contrario:
 - Mostrar "Error: Entrada sintácticamente incorrecta".
 - Leer la siguiente entrada.
 - d. Finalizar el bucle cuando no haya más entrada.
 4. Mostrar "Análisis completado."
 5. Finalizar el proceso principal.

Fin del Proceso Principal

Pseudocódigo

```
SubProceso Analizador_Lexico
    Definir tokens Como Caracter;
    tokens <- "IDENTIFICADOR ENTERO OP_ARIT OP_REL OP_LOG BOOL P_ABRE P_CIERRE";

    Definir entrada Como Caracter;
    Leer entrada;

    Mientras entrada <> FinDeTexto Hacer
        Si entradaEntokens Entonces
            Escribir "Token reconocido: ", entrada;
        Sino
            Escribir "Error: Carácter desconocido";
        FinSi
    FinMientras
FinSubProceso
```



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
SubProceso Analizador_Sintactico
    Definir gramatica Como Caracter;
    gramatica <- "expresion: variable BOOL expresion OP_REL variable expresion OP_LOG
variable expresion OP_ARIT variable P_ABRE expresion P_CIERRE expresion OP_LOG
P_ABRE expresion P_CIERRE; variable: IDENTIFICADOR NUMBER";

    Definir entrada Como Caracter;
    Leer entrada;

    Mientras entrada <> FinDeTexto Hacer
        Si entradaEngramatica Entonces
            Escribir "Entrada sintacticamente correcta: ", entrada;
        Sino
            Escribir "Error: Entrada sintacticamente incorrecta";
        FinSi
    FinMientras
FinSubProceso

Proceso Principal
    Escribir "Iniciando analisis...";
    Analizador_Lexico();
    Analizador_Sintactico();
    Escribir "Analisis completado.";
FinProceso
```

PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II

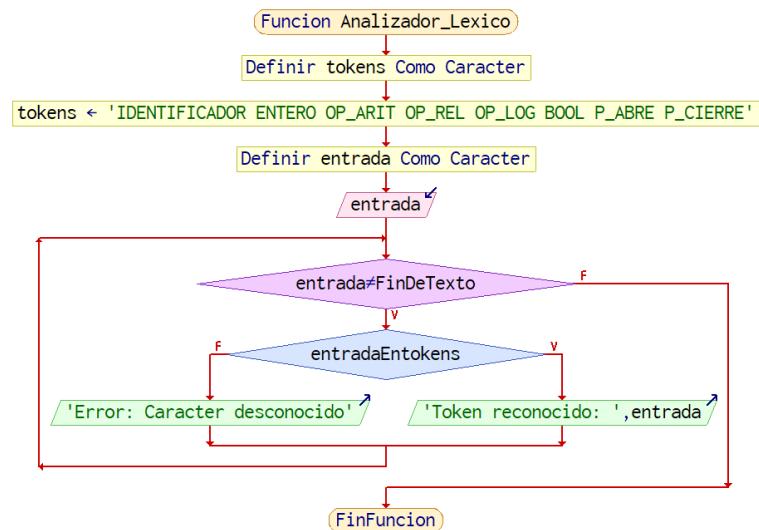
AUTORES:

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

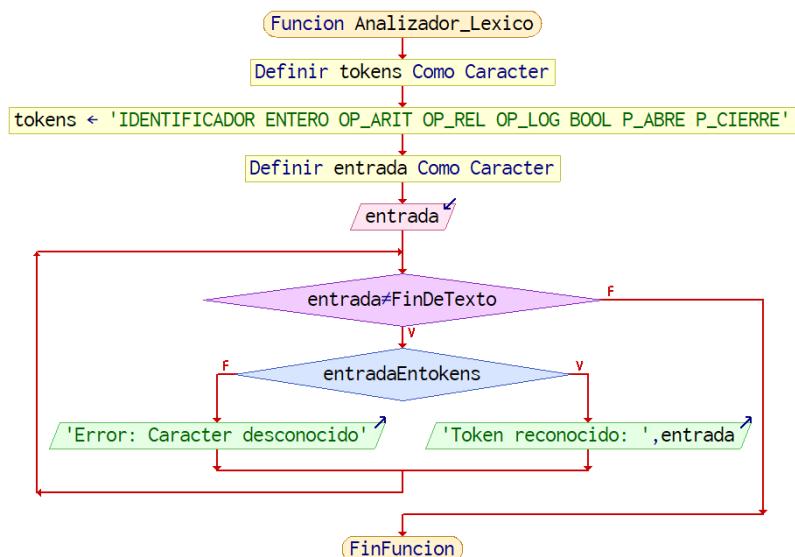
[Handwritten signatures]

Diagrama de flujo

- Analizador léxico



- Analizador sintáctico

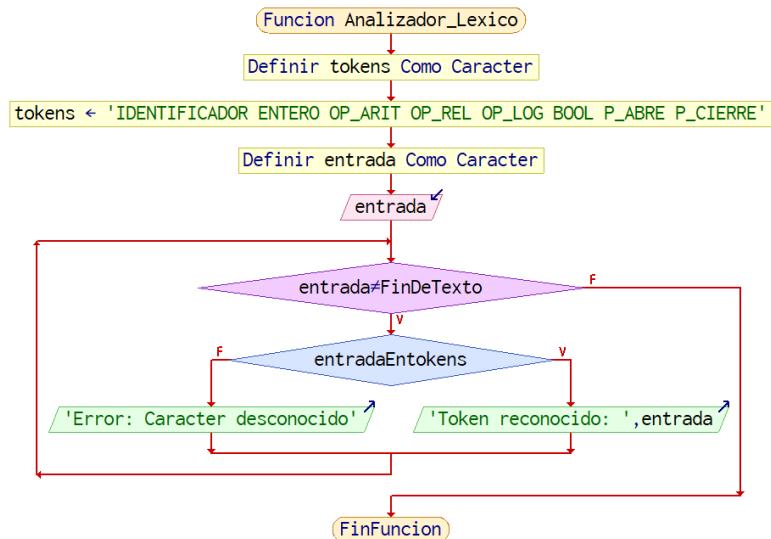


- Proceso principal

PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II

AUTORES:

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



Elaboración de la práctica

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

A continuación, se explica el desarrollo de la práctica:

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

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

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

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

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

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

```
cristhian@CAOH-HP-Laptop:~$ sudo apt-get install bison
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
bison is already the newest version (2:3.8.2+dfsg-1build1).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

En la práctica anterior se creó una carpeta en donde se almacenarían todas las prácticas que se realicen para esta materia. Utilizaremos la misma carpeta, pero dentro de ella se creará otra llamada “practica2” para ahí guardar todos los archivos correspondientes a esta práctica.

```
cristhian@CAOH-HP-Laptop:~/practicaslexyaac$ ls
ejemplo practical practica1LA
cristhian@CAOH-HP-Laptop:~/practicaslexyaac$ mkdir practica2
cristhian@CAOH-HP-Laptop:~/practicaslexyaac$ ls
ejemplo practical practica1LA practica2
cristhian@CAOH-HP-Laptop:~/practicaslexyaac$ cd practica2
cristhian@CAOH-HP-Laptop:~/practicaslexyaac/practica2$ |
```

Para crear los archivos, se puede hacer de dos maneras:

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

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

```
cristhian@CAOH-HP-Laptop:~/practicaslexyaac$ cd practica2
cristhian@CAOH-HP-Laptop:~/practicaslexyaac/practica2$ code .
```

Con Visual Studio Code abierto, se procede a crear un archivo en el cual se especificará la gramática a utilizar. Este archivo tendrá una extensión “.y” y se llamará “reconocedorLogico”.

 reconocedorLogico.y

Los archivos de especificación de gramática se dividen en tres secciones, las cuales se explicarán a continuación:

1.- Definiciones: Aquí se definen macros, funciones y variables que serán utilizadas posteriormente.

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
int yyerror(char *s);
```

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

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

```
extern int yylineno;
}

%token OTHER NUMBER IDENTIFICADOR BOOL OP_LOG OP_REL OP_ARIT P_ABRE P_CIERRE
%type <numero> NUMBER
%type <id> IDENTIFICADOR

%union{
    int numero;
    char id[100];
}
```

En el caso de este ejemplo, primero se incluyen los archivos de cabecera estándar “<stdio.h>” y “<stdlib.h>”, que proporcionan funciones para entrada/salida estándar y gestión de memoria dinámica. Después se declara una función llamada “yylex()”, que será utilizada para obtener tokens del analizador léxico. También se declara una función llamada “yyerror()” para manejar errores sintácticos durante el análisis. La variable “yylineno” se utilizará para rastrear el número de línea actual en el código.

Posteriormente se declaran los tokens que se utilizarán durante el análisis. También se especifica el tipo de valor que se asocia a ciertos tokens, por ejemplo, “NUMBER” se asocia a un tipo llamado “numero”.

Por último, se define una unión, esta es utilizada para almacenar valores de diferentes tipos asociados a los tokens. La unión tiene dos campos:

- int numero: Se utiliza para representar valores numéricos asociados a tokens como “NUMBER”.
- char id[100]: Se utilizar para representar identificadores asociados a tokens como “IDENTIFICADOR”.

2.- Reglas: Aquí se definen las reglas de la gramática.

```
expresion:
    variable
    | BOOL
    | expresion OP_REL variable
    | expresion OP_LOG variable
    | expresion OP_ARIT variable
    | P_ABRE expresion P_CIERRE
    | expresion OP_LOG P_ABRE expresion P_CIERRE
;

variable:
    IDENTIFICADOR
    | NUMBER
```

Para esta gramática se definieron dos reglas:

- expresion: Esta regla define como se construyen expresiones en este lenguaje. Puede tomar varias formas, según las alternativas separadas por |.
- variable: Esta regla describe como se definen las variables en el lenguaje. Una variable puede

**PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II****AUTORES:**

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

ser un identificador o un número.

3.- Funciones adicionales en C: Aquí se definen funciones que serán copiadas tal cual al final del archivo generado por Bison.

```
int yyerror(char *s){  
    printf("<<ERROR SINTACTICO: %s en la linea %d>>\n", s,yylineno);  
    return 0;  
}  
  
int main(int argc, char **argv){  
    yyparse();  
    return 0;  
}
```

En la función main se aceptan dos argumentos: “argc”, que es el número de argumentos en la línea de comandos, y “argv”, que es un arreglo de cadenas que contiene los argumentos en la línea de comandos. La función “yyparse()” se llama en el cuerpo de main. “yyparse()” es una función generada por Yacc o Bison que inicia el proceso de análisis sintáctico.

La función “yyerror()” se utiliza para manejar errores sintácticos que ocurren durante el análisis de un programa en el lenguaje definido por la gramática. Recibe un argumento “s”, que es una cadena de caracteres que proporciona información sobre el error.

El archivo de especificaciones de Flex tiene la misma estructura que el archivo de Bison:

1.- Definiciones: Aquí se definen macros y patrones de expresiones regulares que serán utilizados en las reglas.

```
%{  
    #include <stdio.h>  
    #include <string.h>  
    #include "reconocedorLogico.tab.h"  
    void showError();  
}  
  
%option yylineno  
DIGITO          [0-9]  
LETRA           [a-zA-Z]  
GUIONES         [-_]  
  
IDENTIFICADOR   {LETRA} ({LETRA} | {DIGITO} | {GUIONES}) *  
ENTERO          -?{DIGITO}+
```

Primero se incluyen los archivos de cabecera estándar “<stdio.h>” y “<string.h>”, que proporcionan funciones para entrada/salida estándar y manejo de cadenas de caracteres. También incluye un archivo de encabezado llamado “reconocedorLogico.tab.h”. Este archivo podría contener definiciones o declaraciones necesarias para la interacción con un analizador sintáctico generado por Bison. Además, se declara una función llamada “showError()”.

La línea “%option yylineno” especifica que Flex debe hacer un seguimiento de los números de línea en el archivo fuente. Cada vez que un token se detecta, Flex incrementará automáticamente la variable

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

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

“yylineno” para hacer un seguimiento de la línea actual.

Por último, se definen los patrones que se utilizan para reconocer tokens:

- “DIGITO”, “LETRA” y “GUIONES” representan lo mismo que su nombre indica.
- “IDENTIFICADOR” se usa para reconocer los identificadores en el lenguaje. Comienza con una letra y puede contener letras, dígitos y guiones.
- “ENTERO” reconoce números enteros que pueden ser positivos o negativos. Pueden comenzar con un signo negativo seguido de uno o más dígitos.

2.- Reglas: Se especifican las reglas que definen como se deben de reconocer y tokenizar las secuencias de caracteres de entrada.

```
[ \t\n]      {}  
"("          {return (P_ABRE);}  
")"          {return (P_CIERRE);}  
"+" | "-" | "*" | "/" | "%" {return (OP_ARIT);}  
"==" | "!=" | ">" | ">=" | "<" | "<=" {return (OP_REL);}  
"and" | "AND" | "or" | "OR" {return (OP_LOG);}  
"true" | "TRUE" | "false" | "FALSE" {return (BOOL);}  
{IDENTIFICADOR}           {sscanf(yytext,"%s",yylval.id); return (IDENTIFICADOR);}  
{ENTERO}                 {yylval.numero = atoi(yytext); return (NUMBER);}  
.                         {showError("Carácter desconocido"); return(OTHER);}
```

La primera regla define un patrón que coincide con caracteres de espacio en blanco, tabulaciones y saltos de línea. La acción asociada está vacía, lo que significa que no se realizará ninguna acción.

La segunda y tercera regla definen patrones que coinciden con un paréntesis de apertura y cierre.

La cuarta regla define un patrón que coincide con operadores aritméticos.

La quinta regla define un patrón que coincide con operadores lógicos.

La sexta regla define un patrón que coincide con los valores booleanos “true” y “false”.

La séptima regla utiliza una expresión regular anteriormente definida para reconocer identificadores.

La octava regla también utiliza una expresión regular para reconocer números enteros positivos o negativos.

La novena regla corresponde a un patrón en el cual no se reconoce el carácter ingresado.

3.- Código en C: Contiene código en C que se requiera y que será copiado al final del archivo generado por Flex.

```
void showError(char* msg){  
    printf("<<ERROR LEXICO: \"%s en la linea %d\">>",msg,yylineno);  
}
```

La función “showError()” se usa para mostrar un error cuando ocurre algún problema con el análisis léxico.

Una vez realizado el código necesario, se procede a generar y compilar el analizador léxico y el analizador sintáctico. Para simplificar la ejecución de comandos, se crea un script en un archivo llamado “script.sh”. El script es el siguiente:



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
#!/bin/bash
clear
echo "<Inicio>"
flex -l lexico.l
bison -dv -Wcounterexamples reconocedorLogico.y
gcc -o ReconocedorLogico lex.yy.c reconocedorLogico.tab.c -lfl
./ReconocedorLogico < prueba.txt
```

Lo que se hace es:

1. Limpiar la pantalla de la terminal para hacer más limpia y legible la salida.
2. Imprimir “<Inicio>” para indicar que ya se está ejecutando el script.
3. Se ejecuta el comando de Flex para generar el analizador léxico a partir del archivo “léxico.l”.
4. Se ejecuta el comando de Bison para generar el analizador sintáctico a partir del archivo “reconocedorLogico.y”.
5. Se compilan los archivos generados por Flex y Bison para crear un programa ejecutable llamado “ReconocedorLogico”.
6. Se ejecuta el programa “ReconocedorLogico” y se le da como entrada el archivo “prueba.txt”.

La entrada que se le da al programa es la siguiente:

```
((2+4)>C)AND(D<=C)
```

Y esta es la salida que proporciona:

```
<Inicio>
cristhian@CAOH-HP-Laptop:~/practicaslexyaac/practica2$
```

No muestra nada debido a que la entrada está correctamente escrita.

Ahora se cambiará la entrada para producir un error léxico:

```
((2+4)>C)AND$(D<=C)
```

Y este es el resultado:

```
<Inicio>
<<ERROR LEXICO: "Caracter desconocido en la linea 1">><<ERROR SINTACTICO: syntax error en la linea 1>>
```

Este error se debe a que se agregó un token que no reconoce el analizador, el cual es “\$”.

Para un error sintáctico, se proporcionará la siguiente entrada:

```
((2+4)>C)AND(D<=C))
```

El mismo editor de código ya indica que hay un error ahí, pero vamos a ver la salida del analizador:

```
<Inicio>
<<ERROR SINTACTICO: syntax error en la linea 1>>
```

Este error se debe a que el número de paréntesis de apertura y cierre no coinciden, en este caso se

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

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

añadieron dos paréntesis de cierre de más.

6 BÍTACORA DE INCIDENCIAS		
Fecha	Problema encontrado	Solución
6/10/2023	El analizador sintáctico descendente no funcionó con Flex y Bison.	Se investigó cómo se realiza un analizador sintáctico descendente y se descubrió que se utiliza la herramienta ANTLR para eso.

7 CONCLUSIÓN	
En la práctica en la que se implementaron un analizador sintáctico descendente con ANTLR y un analizador sintáctico ascendente con Flex y Bison, se logró una comprensión más profunda de dos enfoques fundamentales en el análisis sintáctico. Cada enfoque tiene sus ventajas y desafíos, y ambos son ampliamente utilizados en la construcción de compiladores y analizadores para diferentes lenguajes de programación.	
En general, la práctica de implementar tanto un analizador sintáctico descendente como uno ascendente brinda una valiosa experiencia en el diseño, desarrollo y prueba de analizadores sintácticos. Esto es fundamental para comprender cómo funcionan los compiladores y los lenguajes de programación, y cómo se pueden aplicar estos conocimientos en proyectos de desarrollo de software y herramientas de análisis de código.	

8 REFERENCIAS	
Antlr. (s. f.). Antlr4/doc/getting-started.md at Master · antlr/antlr4. GitHub. Consultado el 7 de octubre de 2023: https://github.com/antlr/antlr4/blob/master/doc/getting-started.md	
Andres2a. (s. f.). GitHub - ANDRES2A/Guia-Instalacion-ANTLR4: This guide will show you how to install ANTLR4 a powerful parser generator on all platforms. GitHub. https://github.com/ANDRES2A/Guia-Instalacion-Antlr4	



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

[Handwritten signatures]

Analizador	sintáctico	ascendente.	(s. f.).	Scribd.
https://es.scribd.com/document/545693154/Analizador-Sintactico-Ascendente				
Compilador	diseño	-	analizador	descendente. (s. f.).
https://www.tutorialspoint.com/es/compiler_design/compiler_design_top_down_parser.htm				
Compilador	diseño	-	analizador	descendente. (s. f.).
https://www.tutorialspoint.com/es/compiler_design/compiler_design_top_down_parser.htm				
Reyes, J. V. (2011, March 3). Analizadores Ascendentes - Cartagena99. https://www.cartagena99.com/recursos/tuneapdf/index.php?archivo=alumnos/apuntes/PDL_08_Tema%205_Analisis%20sintactico%20ascendente.pdf				

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

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

[Handwritten signatures]

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

PRACTICA NO.	NOMBRE DE LA PRACTICA	FECHA DE ENTREGA
2	FUNCIONAMIENTO DEL GENERADOR DE CÓDIGO PARA ANALIZADORES SINTÁCTICOS "YACC"	11/10/23

VIDEO: [Link](#)

1	INTRODUCCION
<p>El generador de analizadores sintácticos o parsers YACC (Yet Another Compiler Compiler) es una herramienta esencial para la creación de compiladores y analizadores léxicos y sintácticos personalizados. En esta práctica, se profundizará en detalles sobre el funcionamiento de YACC mediante dos ejercicios prácticos de calculadoras donde una es continuidad de otra y con un último ejercicio donde se utiliza la estructura gramatical del inglés y se estandariza en un compilador LEX-YACC. Estos ejercicios no solo proyectarán los conceptos fundamentales de YACC, sino que también mostrarán a través de la experiencia práctica cómo esta herramienta transforma las especificaciones gramaticales en código ejecutable. A medida que se avance en estos ejercicios, se desglosarán las reglas gramaticales, se definirán tokens y se construirá un analizador sintáctico que pueda procesar expresiones aritméticas y producir resultados significativos.</p> <p>El enfoque de esta práctica no sólo se limitará a la generación de un analizador sintáctico, sino que también se mostrará la importancia de enlazar un analizador léxico al analizador sintáctico generado por YACC. Al conectar estos dos componentes, se logra un análisis completo del código fuente, desde la identificación de elementos léxicos básicos hasta la comprensión de la estructura y organización general del programa. Este enlace garantiza un análisis coherente y efectivo del código fuente y es un paso crucial en la construcción de compiladores e intérpretes robustos.</p>	

2	OBJETIVO
<p>El objetivo de esta práctica es obtener una visión profunda de los conceptos clave detrás de YACC, tales como la definición de reglas gramaticales, manipulación de tokens, generación de código en lenguaje C y la relación entre el análisis sintáctico y el léxico</p>	

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

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

durante el proceso de compilación.

3**FUNDAMENTO**

En el ámbito de los compiladores e intérpretes, conocer el concepto de los generadores de código para analizadores sintácticos es de suma importancia para comprender siquiera la estructura de éstos. Yacc genera analizadores sintácticos y representa la etapa de análisis sintáctico de un compilador/intérprete, etapa que es imprescindible para corroborar que la estructura gramatical de un lenguaje se está respetando.

El análisis sintáctico no es más importante que el semántico o léxico, sino que los tres poseen la misma importancia pues un compilador no puede funcionar totalmente si no cuenta con estas tres etapas de análisis bien definidas y enlazadas.

YACC es la herramienta representante de esta etapa sintáctica, por lo que es importante profundizar en sus conceptos, estructura, características y funcionamiento.

4**MATERIALES NECESARIOS**Hardware:

- CPU: Intel i3 8va generación 2.4 GHz
- RAM: 8GB
- SSD: 128 GB

Software:

- SO: Windows 10 o alguna distribución de Linux (idealmente basada en Debian o Ubuntu, Debian 12 es la utilizada para la actual práctica).
- YACC
- FLEX o LEX
- El compilador GCC (GNU Compiler Collection)
- Algún editor de código ya sea Nano, Vim, Sublime Text, Visual Studio Code o cualquier otro que se prefiera

Conocimientos:

Conocimiento básico del lenguaje de programación C

5**DESARROLLO****HISTORIA DE YACC**

YACC es para los analizadores sintácticos (parsers) lo que LEX es para los analizadores léxicos (scanners). Fue lanzado en la década de 1971 y escrito en los laboratorios Bell para formar parte de versiones de UNIX desde inicios del desarrollo de este sistema operativo.

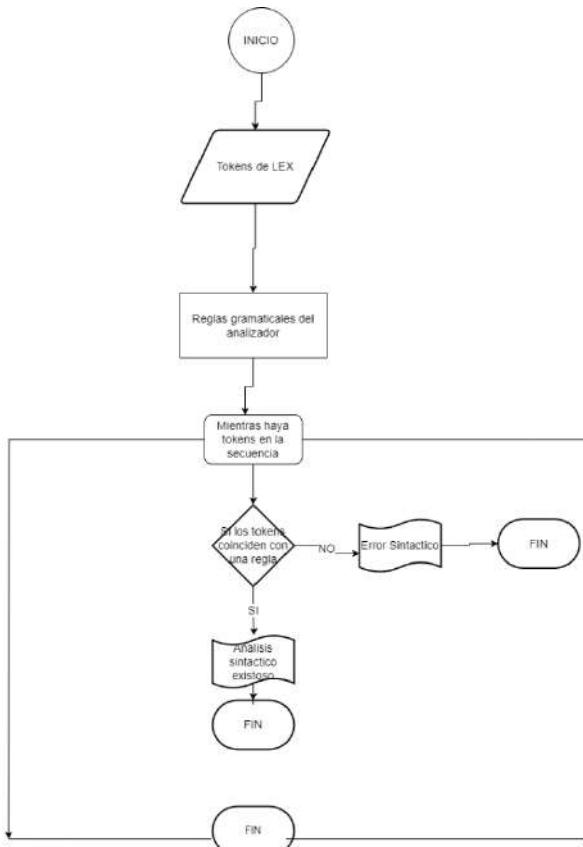
TECNOLÓGICO
NACIONAL DE MÉXICO**INSTITUTO TECNOLÓGICO DE CELAYA****PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II****AUTORES:**

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

Uno de los principales usos de YACC fue el de desarrollar el Compilador Portable de C, abreviado como 'pcc'. Todo esto fue desarrollado por Steven C. Johnson y, de acuerdo con él, YACC no fue originalmente escrito en C, sino es su antecesor B; fue hasta 1973 que se escribió en C.

YACC ayudó en la expansión temprana de UNIX, pues originalmente sólo estaba disponible en este sistema operativo, además de que con el compilador de C 'pcc' también ayudó a aumentar su popularidad. Sin embargo, llegada la década de los 90s, YACC perdió uso cuando programas similares con licencias menos restrictivas y con mayores características comenzaron a salir.

YACC no era de código abierto como tal hasta hace apenas un par de décadas cuando en 2002, Caldera, una de las empresas desarrolladoras de UNIX, puso a disposición del público las fuentes de versiones de UNIX donde YACC estaba incluido. Sin embargo, para ese entonces, Bison ya había ganado mucha mayor popularidad y había reemplazado a YACC hasta en las distribuciones UNIX en las cuales era nativo.

DIAGRAMA DE FLUJO Y PSEUDOCODIGO



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

Entrada: Secuencia de tokens desde Lex

Definir reglas gramaticales en YACC

Mientras haya tokens en la secuencia

Intentar hacer coincidir los tokens con las reglas gramaticales

Si los tokens coinciden con una regla

Realizar la acción asociada con esa regla

Si todos los tokens coinciden con las reglas gramaticales

Retornar "Análisis sintáctico exitoso"

De lo contrario

Retornar "Error sintáctico"

INSTALACIÓN DE YACC Y LEX EN LINUX

YACC es una herramienta antecesora de Bison utilizada durante la década de los 70s y parte de los 80s, mientras que Bison es una herramienta más moderna, de código abierto y mejorada que se creó para superar algunas limitaciones de YACC. Pese a todo, Bison es compatible con YACC y sus formatos/especificaciones, por lo que, para esta práctica, se utilizará YACC mediante el paquete de instalación de Bison en Debian. Para utilizar YACC, simplemente se debe ejecutar el comando 'yacc'.

1. **Instalar el paquete de GCC.** Para instalar el paquete de GCC y poder usar este compilador de C, se debe utilizar el siguiente comando:

```
sudo apt install build-essential
```

Este paquete contiene el compilador GCC, además de otras utilidades requeridas para construir software.

2. **Instalar el generador de scanners FLEX.** Para ello, se debe escribir el siguiente comando:

```
sudo apt-get install flex
```

Con este paquete, se hará uso de Lex/Flex para generar un analizador léxico que, junto con el analizador sintáctico hecho por YACC, funcionen complementariamente.

3. **Instalar el generador de parsers YACC.** Escribir el siguiente comando en la terminal de Debian:



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

sudo apt-get install bison

El paquete se llama bison y, efectivamente, se trata de el generador de analizadores sintácticos de código abierto llamado Bison, sin embargo, Bison es compatible con Yacc y todas sus características por lo que, aquello que se programe o defina en un archivo de Bison, será también compatible con Yacc y viceversa. Además, este paquete de Bison contiene un comando especial llamado 'yacc' que permite ejecutar directamente esta herramienta.

ESTRUCTURA DE YACC

Yacc compila un archivo, el cual contiene las especificaciones de una gramática y la sintaxis que debe adoptar una cadena introducida en base a esta gramática.

Este archivo debe ser de extensión '.y' y tiene una estructura intencionalmente parecida a la de Lex:

```
%{  
Declaraciones  
%}  
Definiciones  
%%  
Producciones  
%%  
Subrutinas de usuario
```

Las secciones de *Declaraciones* y *Subrutinas de usuario* son opcionales, y son escritas en código C estándar que puede ser incluido en el archivo '.c' que se genera al correr Yacc. Las Declaraciones serán incluidas hasta la cabeza del archivo generado y las Subrutinas hasta abajo.

La sección de *Definiciones* es donde se pueden configurar varias características del analizador sintáctico, como variables globales que comuniquen al analizador sintáctico con el léxico, tokens, precedencia de los operadores, etcétera; la sección de *Producciones* es obligatoria y la más importante, pues es en donde se especifican las reglas gramaticales.

**PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II****AUTORES:**

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

EJERCICIO 1. CALCULADORA ARITMÉTICA DE SUMAS Y RESTAS

Cabe aclarar que los siguientes ejercicios son extraídos del libro [4], cuyo autor es Tom Niemann, sin embargo, el equipo logra entender los conceptos y funcionalidades del ejemplo, así como la estructura del código.

1. **Posicionarse en una carpeta exclusiva para el compilador.** En este ejemplo, se creó una carpeta llamada "calc1".
2. **Crear el archivo '.l' para el analizador léxico.** Este archivo será ejecutado por LEX, el cual evaluará si cadenas introducidas a la consola son o no pertenecientes a cierta gramática y, si lo son, clasificar de qué tipo de símbolos está conformada.

Su código es el siguiente:

```
GNU nano 7.2
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%
%%
[0-9]+ {
    yylval = atoi(yytext);
    return INTEGER;
}
[-+\n] return *yytext;
[ \t] ; /* skip whitespace */
. yyerror("invalid character");
%
int yywrap(void) {
    return 1;
}
```

Stdlib.h es la librería estándar de C y "y.tab.h" es simplemente el código del encabezado del analizador sintáctico que contiene analizadores de símbolos.

Se debe recordar que al ejecutar un archivo 'y' con YACC, éste genera dos archivos, uno de los cuales es "y.tab.c" que contiene todo el código de C para el analizador sintáctico; y el encabezado "y.tab.h", el cual es el encargado de proporcionar definiciones de constantes simbólicas que representan los tokens y símbolos no terminales dentro del analizador sintáctico.

3. **Crear el archivo 'y' para el analizador sintáctico.** Este código debe contener las especificaciones y restricciones gramaticales que tendrá el lenguaje compilado. Debe definir la sintaxis y la estructura del código. En el caso de la calculadora, se debe especificar que el operador (sea suma o resta) debe ir entre los dos identificadores o números para que tenga una sintaxis válida y se puedan realizar las operaciones.



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

Su código es el siguiente:

```
GNU nano 7.2
|{
  #include <stdio.h>
  int yylex(void);
  void yyerror(char *);
}
%token INTEGER
%%
program:
  program expr '\n' { printf("%d\n", $2); }
  |
  ;
expr:
  INTEGER { $$ = $1; }
  | expr '+' expr { $$ = $1 + $3; }
  | expr '-' expr { $$ = $1 - $3; }
  ;
%%
void yyerror(char *s) {
  fprintf(stderr, "%s\n", s);
}
int main(void) {
  yyparse();
  return 0;
}|
```

La línea 6 del código, en la sección de Declaraciones, es un token llamado INTEGER, el cual establece una correspondencia entre un nombre simbólico y un token que puede ser reconocido por un analizador léxico. Si se mira el código del archivo '.l' de Lex, se definió que todo número del 0 al 9 (y sus infinitas combinaciones) regrese el valor ingresado bajo el token llamado INTEGER, es de aquí de donde se recupera el nombre de ese token para utilizarlo en el código del analizador sintáctico.

La línea 8 es un no-terminal llamado *program*, que puede consistir en cero elementos o en una secuencia de *expr* (expresiones) seguidas por un salto de línea.

La línea 12 es otro no-terminal llamado *expr* que puede tener tres formas gramaticales distintas: Una donde solo es un número entero, otra donde se realiza una suma (bajo la regla gramatical de que el operador siempre va entre los números) y la última, donde se realiza una resta.

La parte de la sección de Producciones recuerda mucho a la notación BNF, es aquí donde se puede observar el estándar de esta notación y la practicidad que tiene.

**PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II****AUTORES:**

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

4. **Ejecutar archivo .l con Lex.** Esto generará un archivo '.c' que trabajará en conjunto con el archivo generado por YACC de extensión '.c' para construir al compilador.

```
root@DESKTOP-NSTQ91F:/home/miguel/calc1# lex scanner.l
root@DESKTOP-NSTQ91F:/home/miguel/calc1# ls -l
total 52
-rw-r--r-- 1 root root 44538 Oct  9 00:56 lex.yy.c
-rw-r--r-- 1 root root    344 Oct  9 00:34 parser.y
-rw-r--r-- 1 root root    243 Oct  9 00:05 scanner.l
```

5. **Ejecutar archivo .y con YACC.** Muy similar al paso anterior, YACC generará un archivo '.c' en donde contiene todas las instrucciones de trabajo en C y adicionalmente un archivo '.h' para vincular el analizador léxico ya escrito con el recientemente creado analizador sintáctico.

```
root@DESKTOP-NSTQ91F:/home/miguel/calc1# yacc -d parser.y
parser.y:17 parser name defined to default :"parse"
conflicts: 4 shift/reduce
root@DESKTOP-NSTQ91F:/home/miguel/calc1# ls -l
total 92
-rw-r--r-- 1 root root 44538 Oct  9 00:56 lex.yy.c
-rw-r--r-- 1 root root    344 Oct  9 00:34 parser.y
-rw-r--r-- 1 root root    243 Oct  9 00:05 scanner.l
-rw-r--r-- 1 root root 32113 Oct  9 00:59 y.tab.c
-rw-r--r-- 1 root root  6154 Oct  9 00:59 y.tab.h
```

El parámetro -d en el comando yacc hace que se genere el tan necesario archivo "y.tab.h".

6. **Vincular ambos archivos '.c' y hacer el compilador.** Utilizar el compilador de C para GNU/Unix llamado 'gcc' y vincular ambos archivos '.c' recién generados.

```
root@DESKTOP-NSTQ91F:/home/miguel/calc1# gcc lex.yy.c y.tab.c
root@DESKTOP-NSTQ91F:/home/miguel/calc1# ls -l
total 120
-rwxr-xr-x 1 root root 27816 Oct  9 01:04 a.out
-rw-r--r-- 1 root root 44538 Oct  9 00:56 lex.yy.c
-rw-r--r-- 1 root root    344 Oct  9 00:34 parser.y
-rw-r--r-- 1 root root    243 Oct  9 00:05 scanner.l
-rw-r--r-- 1 root root 32113 Oct  9 00:59 y.tab.c
-rw-r--r-- 1 root root  6154 Oct  9 00:59 y.tab.h
```

Cuando se haga esto, se generará un archivo de extensión .out el cual es el compilador en sí.

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

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

7. **Ejecutar el archivo a.out.** Esto no requiere de un comando en especial, solo se necesita escribir la ruta del archivo y escribir su nombre y con eso es suficiente para ejecutar el compilador/calculadora aritmética de suma y resta.

El resultado se puede ver como el siguiente:

```
root@DESKTOP-NSTQ91F:/home/miguel/calc1# ./a.out
1+2
3
22+1
23
22-90
-68
```

EJERCICIO 2. Calculadora Aritmética Mejorada

El siguiente ejercicio es la continuación del ejercicio anterior. Este es una mejora de la calculadora anterior en la cual se agregan nuevas funcionalidades como las operaciones aritméticas de multiplicación y división, además de la asignación de variables e inclusión de paréntesis para anular la precedencia de operadores.

La siguiente imagen es una muestra de entradas del usuario y las salidas que daría el compilador:

```
user: 3 * (4 + 5)
calc: 27
user: x = 3 * (4 + 5)
user: y = 5
user: x
calc: 27
user: y
calc: 5
user: x + 2*y
calc: 37
```

Donde ‘user’ son las entradas de usuario y ‘calc’ son las salidas dadas por el compilador/calculadora.

1. **Crear el archivo ‘l’ en una nueva carpeta para la calculadora mejorada.** La carpeta creada para este nuevo ejercicio se llamó ‘lxyc’ y ahí se creó el archive ‘scanner.l’ con el editor nano. Su código es el siguiente:

**PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II****AUTORES:**

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

```
GNU nano 7.2
},{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
}
%%
/* variables */
[a-z] {
    yylval = *yytext - 'a';
    return VARIABLE;
}
/* integers */
[0-9]+ {
    yylval = atoi(yytext);
    return INTEGER;
}
/* operators */
[-+()*/\n] { return *yytext; }
/* skip whitespace */
[ \t];
/* anything else is an error */
. yyerror("invalid character");
%%
int yywrap(void) {
    return 1;
}
```

2. **Crear el archivo con extensión 'y'.** Este archivo será ejecutado por yacc, su nombre será 'parser.y' y su código es el siguiente:

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
%
{
void yyerror(char *);
int yylex(void);
int sym[26];
}
%%
program:
program statement '\n'
|
;
statement:
expr                  { printf("%d\n", $1); }
| VARIABLE '=' expr { sym[$1] = $3; }
;
expr:
INTEGER
```

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

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

```
| VARIABLE      { $$ = sym[$1]; }
| expr '+' expr { $$ = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr { $$ = $1 / $3; }
| '(' expr ')' { $$ = $2; }
;
%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}
int main(void) {
    yyparse();
    return 0;
}
```

Como dice textualmente Niemann [4] en su calculadora ejemplo, “Los tokens para INTEGER y VARIABLE son utilizados por YACC para crear #defines en el encabezado y.tab.h para que sean utilizados por LEX. A las declaraciones de estos tokens les siguen las definiciones de los operadores aritméticos. Se puede especificar %left para hacer asociación a la izquierda o %right para asociación a la derecha. La última definición listada tiene la mayor precedencia (prioridad). Por lo tanto, la multiplicación y la división tiene mayor precedencia que la suma y la resta. Los cuatro operadores tienen asociación a la izquierda.”

Para la parte de la declaración (statement) tiene la posibilidad de asignarle a una variable (Cualquier letra de la ‘a’ a la ‘z’ minúscula, esto definido en LEX) con el símbolo ‘=’ seguido de una expresión, la cual puede ser ya sea un INTEGER (0,1,2,...,n) o una operación aritmética como la suma de dos enteros o la resta.

La forma de declarar las expresiones para la multiplicación y la división es exactamente la misma que para la suma y la resta, solo que se usan los operadores ‘*’ y ‘/’

Resultados



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
root@DESKTOP-NSTQ91F:/home/miguel/lxyc# ./a.out
4 - 2
2
r = 3/1
r
3
y = 50
y
50
z = r*y
z
150
```

- La primera línea es una expresión, en la que se realiza una operación de resta.
- La segunda línea es el resultado de la resta definida en la línea anterior.
- En la tercera línea se “almacena” en una variable llamada ‘r’ la expresión ‘3/1’
- En la cuarta línea si se teclea ‘r’, devolverá el resultado de la operación de esa expresión.
- En las líneas posteriores se muestran más ejemplos del funcionamiento de esta calculadora.



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

[Handwritten signatures]

EJERCICIO 3. Reconocedor de Oraciones Simples y Compuestas en Inglés

El siguiente ejercicio fue extraído directamente del sitio web de O'Reilly Media, Inc. [6] y fue desarrollado por John Levine, Doug Brown, Tony Mason en su libro titulado *lex & yacc, 2nd Edition*.

El programa consiste en un analizador léxico-sintáctico que analice palabras y que, en base a la gramática estándar del inglés, comience a identificar si la cadena introducida es una oración simple o compuesta; si cierta palabra es un pronombre, sustantivo, verbo, objeto, adverbio, adjetivo, etcétera.

Este análisis no es de ninguna manera un análisis completo del inglés, pues la gramática inglesa es mucho más compleja de lo que se puede presentar en este ejercicio, tan solo la ambigüedad cotidiana que presenta este lenguaje natural es suficiente para recordar lo complejo que puede ser el idioma.

1. **Definir el léxico del programa en un archivo '.l'.** En este archivo, se deben definir los componentes del idioma inglés como palabras clave, identificadores, números o símbolos relevantes para el análisis léxico. Su nombre es 'scanner.l' y se encuentra en una carpeta llamada 'englishComp'.

```
%{  
/*  
 * We now build a lexical analyzer to be used by a  
higher-level parser.  
 */  
  
#include "y.tab.h"      /* token codes from the parser */  
  
#define LOOKUP 0      /* default - not a defined word  
type. */  
  
int state;  
  
%}  
  
%%  
  
\n      { state = LOOKUP; }
```



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
\.\n  {      state = LOOKUP;\n      return 0; /* end of sentence */\n  }\n\n^verb { state = VERB; }\n^adj  { state = ADJECTIVE; }\n^adv  { state = ADVERB; }\n^noun { state = NOUN; }\n^prep { state = PREPOSITION; }\n^pron { state = PRONOUN; }\n^conj { state = CONJUNCTION; }\n\n[a-zA-Z]+\n{\n    if(state != LOOKUP) {\n        add_word(state, yytext);\n    } else {\n        switch(lookup_word(yytext)) {\n            case VERB:\n                return(VERB);\n            case ADJECTIVE:\n                return(ADJECTIVE);\n            case ADVERB:\n                return(ADVERB);\n            case NOUN:\n                return(NOUN);\n            case PREPOSITION:\n                return(PREPOSITION);\n            case PRONOUN:\n                return(PRONOUN);\n            case CONJUNCTION:\n                return(CONJUNCTION);\n            default:\n                printf("%s: don't recognize\n", yytext);\n                /* don't return, just ignore it */\n        }\n    }\n}\n.\n;
```



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

%%

```
/* define a linked list of words and types */
struct word {
    char *word_name;
    int word_type;
    struct word *next;
};

struct word *word_list; /* first element in word list */

extern void *malloc() ;

int
add_word(int type, char *word)
{
    struct word *wp;

    if(lookup_word(word) != LOOKUP) {
        printf("!!! warning: word %s already defined\n", word);
        return 0;
    }

    /* word not there, allocate a new entry and link it
on the list */

    wp = (struct word *) malloc(sizeof(struct word));

    wp->next = word_list;

    /* have to copy the word itself as well */

    wp->word_name = (char *) malloc(strlen(word)+1);
    strcpy(wp->word_name, word);
    wp->word_type = type;
    word_list = wp;
    return 1; /* it worked */
}
```

**PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II****AUTORES:**

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

```
int
lookup_word(char *word)
{
    struct word *wp = word_list;

    /* search down the list looking for the word */
    for(; wp; wp = wp->next) {
        if(strcmp(wp->word_name, word) == 0)
            return wp->word_type;
    }

    return LOOKUP;                                /* not
found */
}
int yywrap()
{
    return 1;
// Indica que no hay más entradas para analizar
}
```

Es importante resaltar que los valores de los tokens, tales como pronombres, adjetivos, adverbios, etcétera serán definidos por el usuario, es decir, cuando el usuario defina un verbo, lo hará bajo la sintaxis **verb jump scream steals**, de esta manera, se estarían introduciendo al léxico del compilador tres verbos en tiempo real al estarse ejecutando el programa. Esto se hace gracias a una tabla de símbolos que almacena estos valores introducidos por el usuario, es una estructura de datos muy común en aplicaciones de LEX y YACC.

Hacer uso de una tabla de símbolos cambia significativamente el analizador léxico. En lugar de colocar patrones separados en el analizador léxico para cada palabra que coincide, se tiene un solo patrón que coincide con cualquier palabra (En el alfabeto latino-inglés) y se consulta la tabla de símbolos para determinar qué parte de la oración se ha encontrado. Los nombres de las partes de la oración o input (sustantivo, verbo, etc.) ahora son 'palabras reservadas', ya que introducen una línea de declaración.

Además, al final del código se pueden encontrar las rutinas de mantenimiento llamadas **add_word()** y **lookup_word()** donde, respectivamente, una añade una

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

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

palabra a la tabla de símbolos y la otra busca una palabra o valor existente en la tabla de símbolos para hacer uso de ella.

2. **Definir las reglas sintácticas y gramaticales del programa en un archivo '.y'.** Este paso es clave para poder escribir oraciones gramaticalmente correctas en el programa. Mientras que el analizador léxico se encarga de registrar, consultar y clasificar palabras dentro de su tabla de símbolos, el analizador sintáctico se encarga de corroborar que su estructura sea válida y aceptable dentro de los parámetros establecidos, los cuales fueron basados en estructuras estándares del inglés. El nombre del archivo es 'parser.y' y se encuentra dentro de la misma carpeta que el analizador léxico. Su código es el siguiente:

```
%{  
#include <stdio.h>  
%}  
  
%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION  
CONJUNCTION  
  
%%  
  
sentence: simple_sentence { printf("Parsed a simple  
sentence.\n"); }  
| compound_sentence { printf("Parsed a compound  
sentence.\n"); }  
;  
  
simple_sentence: subject verb object  
| subject verb object prep_phrase  
;  
  
compound_sentence: simple_sentence CONJUNCTION simple_sentence  
simple_sentence  
| compound_sentence CONJUNCTION simple_sentence  
;  
  
subject: NOUN
```

**PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II****AUTORES:**

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

```
|      PRONOUN
|      ADJECTIVE subject
;

verb:      VERB
|      ADVERB VERB
|      verb VERB
;

object:      NOUN
|      ADJECTIVE object
;

prep_phrase:      PREPOSITION NOUN
;

%%

extern FILE *yyin;

main()
{
    do
    {
        yyparse();
    }
    while(!feof(yyin));
}

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

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

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

Nótese que los tokens definidos en este código tienen el mismo nombre que los definidos en el archivo de Lex, esto es necesario para que se pueda establecer una relación entre estos tokens entre analizador léxico y sintáctico mediante el archivo encabezado 'y.tab.h'

El analizador sintáctico básicamente analizará si una cadena introducida por el usuario es una sentencia simple o compuesta. Para ser una sentencia simple debe tener principalmente la estructura **sujeto verbo objeto**; para que sea una sentencia compuesta, su estructura debe principalmente ser **frase_simple conjunción frase_simple**.

El código mostrado de Yacc está aún incompleto, solo puede identificar frases simples, pues aún no tiene definidos los patrones para Conjunción, Preposición, Adjetivo, Adverbio y Pronombre; esto es debido a lo comentado anteriormente, la complejidad y ambigüedad del inglés. Un analizador de oraciones en inglés es muchísimo más complejo de lo que se presenta con este ejemplo, pero este ejercicio es simple, con fines de mostrar el funcionamiento de Yacc en otra aplicación que no sea sólo una calculadora.

3. **Ejecutar ambos archivos con lex y yacc.** Para generar los archivos en lenguaje C de los analizadores léxico y sintáctico, se deben ejecutar los siguientes comandos (ya vistos en el primer ejemplo):

```
miguel@DESKTOP-NSTQ91F:~/lex/englishComp$ lex scanner.l
miguel@DESKTOP-NSTQ91F:~/lex/englishComp$ yacc -d parser.y
miguel@DESKTOP-NSTQ91F:~/Lex/englishComp$ ls -l
total 100
-rw-r--r-- 1 miguel miguel 48303 Oct  9 18:45 lex.yy.c
-rw-r--r-- 1 miguel miguel   889 Oct  9 18:45 parser.y
-rw-r--r-- 1 miguel miguel  2444 Oct  9 18:42 scanner.l
-rw-r--r-- 1 miguel miguel 33783 Oct  9 18:46 y.tab.c
-rw-r--r-- 1 miguel miguel  6523 Oct  9 18:46 y.tab.h
```

Una vez generados estos archivos, ya se pueden enlazar, como se muestra en el siguiente paso.

4. Vincular los archivos de Lex y Yacc con extensión '.c'. Utilizar el comando gcc para hacer esa vinculación y así generar el archivo ejecutable de los analizadores.

```
miguel@DESKTOP-NSTQ91F:~/lex/englishComp$ gcc lex.yy.c y.tab.c
```

Ahora, se puede visualizar el archivo *a.out*, el cual, al ser ejecutado, permite utilizar los analizadores recién definidos.



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
miguel@DESKTOP-NSTQ91F:~/lex/englishComp$ ls -l
total 132
-rwxr-xr-x 1 miguel miguel 32192 Oct  9 18:51 a.out
```

Resultados

Como se mencionó en el punto número dos de este ejercicio, es el usuario quien tiene que introducir y definir los valores para los tokens de verbo, sustantivo, objeto, etc, para ello, primero se definirán tres verbos y seis sustantivos:

```
miguel@DESKTOP-NSTQ91F:~/lex/englishComp$ ./a.out
verb sings teaches watches
noun Marie RGG Gustavito music automatons TV
```

En la imagen anterior, se utiliza las palabras reservadas (definidas en el archivo lex) **verb** y **noun** para definir tres verbos y seis sustantivos y añadirlos a la tabla de símbolos.

Cuando se introduce una cadena con una sintaxis aceptada para el analizador sintáctico, se indicará que se ha analizado una sentencia simple (simple porque es la única definida, las compuestas aún no funcionan):

```
Marie sings music
Parsed a simple sentence.
RGG teaches automatons
Parsed a simple sentence.
Gustavito watches TV
Parsed a simple sentence.
```

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

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

BÍTACORA DE INCIDENCIAS		
FECHA	PROBLEMA ENCONTRADO	SOLUCIÓN
6/10/2023	Un miembro del equipo no lograba comprender la diferencia entre YACC y Bison y no quería instalar el paquete de Bison en Debian para usar Yacc.	Dicho miembro se instruyó con libros digitales sobre la historia de Yacc y su evolución a Bison, y comprendió las diferencias entre ambos. Entendió que para usar Yacc de una manera sencilla en Linux, es recomendable hacerlo a través del paquete de Bison.
9/10/2023	Múltiples declaraciones del método main en el tercer ejemplo de esta práctica. Había un método main en el archivo 'scanner.l' (LEX) y en el de 'paser.y' (YACC)	Se eliminó el método main del archivo 'scanner.l'.
9/10/2023	Hacía falta definir la función yywrap() en el archivo 'scanner.l', y el analizador léxico 'lex.yy.c' tenía referenciada en su código a esa función, pero no existía, por lo que no se ejecutaba el archivo.	Se definió la función yywrap() al final del archivo 'scanner.l'

CONCLUSIÓN	
Esta práctica nos muestra un funcionamiento práctico de YACC, un generador de analizadores sintácticos y la relación que tiene con LEX, un generador de analizadores léxicos. A través de tres ejemplos sencillos se exponen los componentes que Yacc utiliza para generar los analizadores como la declaración de expresiones/patrones, su estructura intencionalmente parecida a la de Lex dividida en secciones, el uso de tokens, etcétera. También se menciona su historia y su relación con Bison, su predecesor más popular hasta ahora. Gracias a esta práctica, el equipo pudo comprender a mayor profundidad la etapa del análisis sintáctico de un compilador, lo que incluye su importancia para definir reglas gramaticales y la sintaxis de un lenguaje. Además, también se logró comprender la estrecha relación complementaria que existe entre la etapa de análisis léxico con el sintáctico mediante la vinculación con el comando 'gcc' de los analizadores léxico y semántico generados por Lex y Yacc.	

REFERENCIAS	
[1] Bison 3.8.1. (s. f.). https://www.gnu.org/software/bison/manual/bison.html#Yacc	



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

[2] Kaplarevic, V. (2023). How to install GCC Compiler on Ubuntu. Knowledge Base by

phoenixNAP. <https://phoenixnap.com/kb/install-gcc-ubuntu>

[3] Johnson, M. (2005). Introduction to yacc and bison. Disponible en:

<https://edoras.sdsu.edu/doc/yacc-intro.pdf>

[4] Niemann, T. (). *Lex & Yacc Tutorial*. Disponible en:

<https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf>

[5] Johnson, Stephen C. [1975]. *Yacc: Yet Another Compiler Compiler*. Computing Science

Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey. A PDF version is

available at ePaperPress.

[6] Levine, J. (s. f.). Lex & YaCC, 2nd Edition. O'Reilly Online Learning.

<https://www.oreilly.com/library/view/lex-yacc/9781565920002/ch01.html>

[7] Mahesh Huddar. (2022, 27 mayo). How to Compile & Run LEX and YACC Programs on

UBUNTU by Dr. Mahesh Huddar [Vídeo]. YouTube. <https://www.youtube.com/watch?v=IffXq-Mhf>

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

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

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

PRACTICA NO.	NOMBRE DE LA PRACTICA	FECHA DE ENTREGA
3	FUNCIONAMIENTO DEL GENERADOR DE CODIGO PARA ANALIZADORES SINTACTICOS "BISON"	11/10/23

VIDEO: [Link](#)

1	INTRODUCCION
<p>En la presente practica se abordara en el rubro del desarrollo de compiladores una de las herramientas más conocidas que facilitan la tarea de la creación de analizadores sintácticos eficientes. Dicha herramienta es BISON que es una alternativa de código abierto de YACC (aunque en fechas recientes ya también lo es), la cual es ampliamente utilizada tanto de forma académica como profesional.</p> <p>Cabe recordar que dicha herramienta está enfocada en la etapa del análisis sintáctico de un compilador dicha etapa desempeña un papel crucial en la comprensión y manipulación de la estructura de los programas informáticos que se han de realizar. Como se mencionó anteriormente, con la ayuda de Bison se podrá automatizar dicho proceso, permitiendo al desarrollador el construir analizadores sintácticos eficientes a partir de especificaciones gramaticales dadas.</p> <p>Se mostrara el funcionamiento de dicha herramienta en base a algunos ejercicios planteados junto con su respectiva documentación, algoritmo, pseudocódigo, diagramas de flujo y el código explicado.</p>	

2	OBJETIVO
<p>Mediante este escrito/practica se busca demostrar y comprender el funcionamiento de la herramienta BISON revisando sus características, funcionalidades instalación, y cuál es su manera de operación es decir que entrada recibirá por parte del programador y que salida se espera que tenga dicho proceso. También se busca plasmar que tanto YACC (practica anterior) y BISON son muy similares dado su origen pero este último es decir BISON presenta ciertas mejoras respecto a su programa base (es decir YACC).</p>	

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

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

3

FUNDAMENTO

Bison es un generador de código que toma una especificación gramatical formal (concretamente una gramática libre de contexto o GLC) como entrada y genera automáticamente un analizador sintáctico en lenguaje C. Dicha especiación gramatical será la encargada de definir las reglas sintácticas y la estructura de un lenguaje o formato de entrada específico. La importancia de bison radica en que se puede crear analizadores sintácticos precisos y muy eficientes sin la necesidad de escribir todo el código implicado de forma manual por el diseñador.

El conocimiento de tanto ya sea Bison o YACC se vuelve esencial para comprender el funcionamiento de los compiladores y analizadores en la práctica.

Mediante el uso de bison se permite abordar proyectos o prácticas relacionadas con el procesamiento de lenguajes de programación de una manera más efectiva y accesible resultando en una mejor comprensión de la estructura sintáctica de los programas y una mayor capacidad para construir herramientas de análisis de software de forma personalizada.

Al hablar sobre un generador de código para el análisis sintáctico se hace mucha alusión a las gramáticas libres de contexto por lo cual el conocimiento de esta se vuelve muy importante ya que dichas son la base de la mayoría de los lenguajes de programación debido a que estos mismos deben contar con una sintaxis predecible y precisa para facilitar su uso por diversos programadores sin la necesidad de que estos sean expertos en el lenguaje y su estructura. Los lenguajes formales (en la cual se utilizan las GLC) son necesarios para crear tanto lenguajes de programación o notaciones científicas ya que a diferencia de los lenguajes naturales, presentan poca o nula ambigüedad, siendo altamente importante para una comprensión única tanto por parte de cualquier programador como también por parte del compilador.

La estructura de las instrucciones de bison es la misma que YACC, a grandes rasgos esta está compuesta de:

- **Definición de tokens:** Bison comienza con esta, los tokens también se conocen como símbolos terminales que se usan en la gramática y que el analizador sintáctico debe reconocer. Dichos se definen usando la directiva '`%token`' y asignan un nombre simbólico a cada uno. **Ejemplos :** `%token IF`, `%token ELSE`, `%token IDENTIFIER`, `%token NUMBER`.
- **Reglas de producción:** Estas especifican la gramática del lenguaje que el analizador sintáctico deberá de analizar. Dichas reglas indican como se construyen las estructuras sintácticas a partir de los tokens y otros símbolos no terminales. Cada

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

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

una de estas consta de un símbolo no terminal seguido de (:) y una secuencia de símbolos terminales y no terminales.

- **Acciones semánticas:** En las reglas de producción se pueden incluir acciones semánticas que se ejecutan cuando se reconoce una estructura sintáctica específica. Estas pueden contener código en el lenguaje de destino (generalmente C) que realiza tareas relacionadas con la semántica del lenguajes. Esto escapa de los alcances de la práctica pero igual se menciona.
- **Directivas y configuración:** Se permite configurar el comportamiento del analizador sintáctico mediante directivas especiales. Estas pueden influir en aspectos como la generación de código, manejo de errores y otras características específicas.
Ejemplos: %start program, %error-verbose.

4

MATERIALES NECESARIOSHardware:

- CPU: Intel i3 8va generación 2.4 GHz
- RAM: 8GB
- SSD: 128 GB

Software:

- SO: Windows 10 o alguna distribución de Linux (idealmente basada en Debian o Ubuntu).
- BISON
- FLEX
- El compilador GCC (GNU Compiler Collection)
- Algún editor de código ya sea Nano, Vim, Sublime Text, Visual Studio Code (aunque este mas bien es un IDE pero igualmente servirá) o cualquier otro de nuestra preferencia

Conocimientos:

- Conocimiento básico del lenguaje del programación de C

**PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II****AUTORES:**

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

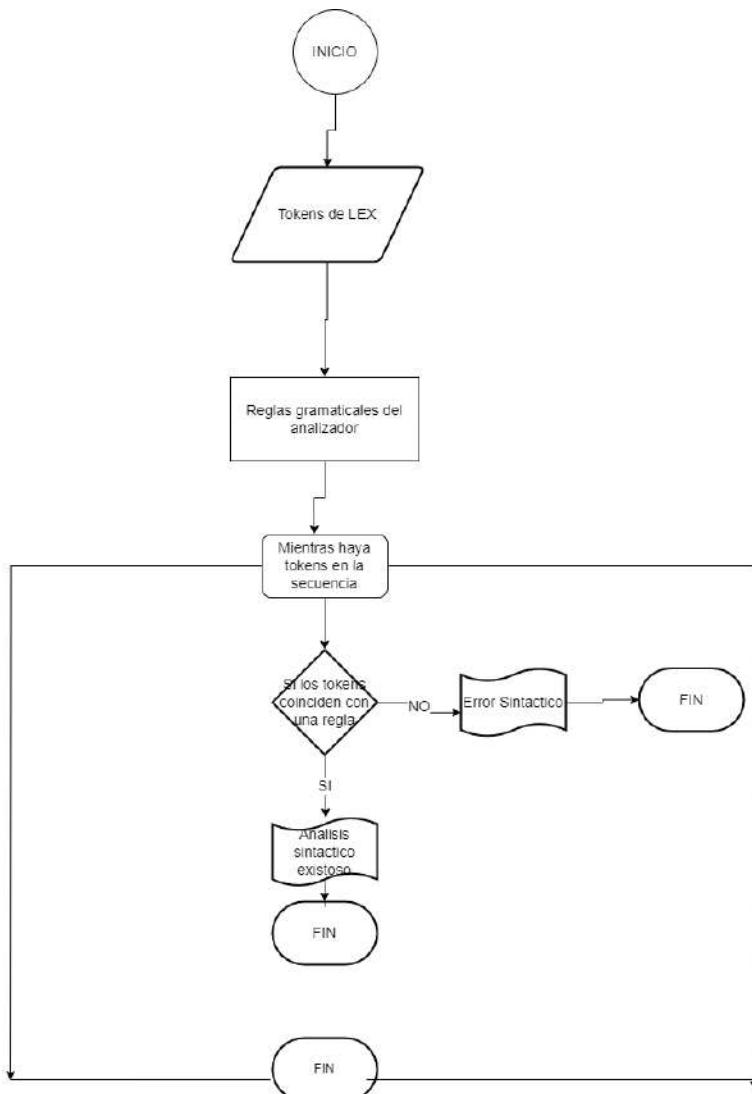
[Handwritten signatures]

5

DESARROLLO

Antes de entrar en detalle a la práctica primero se mostrara a continuación la serie de pasos o proceso que se realizara para utilizar BISON, que recordemos no variara demasiado respecto a YACC.

Primero se mostrara el Diagrama de flujo:





TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

[Handwritten signatures]

Y a continuación el pseudocódigo del algoritmo a emplear:

Entrada: Secuencia de tokens desde Lex

Definir reglas gramaticales en Bison

Mientras haya tokens en la secuencia

Intentar hacer coincidir los tokens con las reglas gramaticales

Si los tokens coinciden con una regla

Realizar la acción asociada con esa regla

Si todos los tokens coinciden con las reglas gramaticales

Retornar "Análisis sintáctico exitoso"

De lo contrario

Retornar "Error sintáctico"

Una vez visto lo anterior se puede entrar en detalle con los ejemplos de uso, los cuales serán dos que han de mostrar a grandes rasgos la funcionalidad y características de dicha herramienta.

Pero antes de iniciar a usar BISON primero se comenzara instalando las dependencias o software necesario para emplear dicho, para ello se utilizará la distribución de Linux llamada PopOS misma que esta basada en Ubuntu (que a su vez esta basada en Debian), pero se podrá utilizar cualquier otra preferentemente basada en Debian o Ubuntu pero también se puede emplear cualquier otra, las únicas diferencias serán relacionadas a los comandos que se utilizaran ya sea para la instalación del software o para la ejecución del compilador GCC pero en esencia el manejo de BISON se mantiene igual. Igual se podría emplear Windows pero se decidió el utilizar Linux ya que dicho es mas empleado a la hora de realizar dichas tareas.

Planteado lo anterior se comenzara con la instalación de BISON:

- 1) Primero deberemos entrar en la terminal y asegurarnos de encontrarnos con usuario ROOT.

```
root@pop-os: /home/agoca
[agoca@pop-os] ~
$ sudo su
[sudo] contraseña para agoca:
root@pop-os:/home/agoca#
```



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

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

- 2) Para realizar la descarga e instalación de bison escribimos el siguiente comando dentro de la terminal: **sudo apt-get install bison** y esperamos su instalación.

```
root@pop-os:/home/agoca# sudo su
[sudo] contraseña para agoca:
root@pop-os:/home/agoca# sudo apt-get install bison
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
Paquetes sugeridos:
  bison-doc
Se instalarán los siguientes paquetes NUEVOS:
  bison
0 actualizados, 1 nuevos se instalarán, 0 para eliminar y 0 no actualizados.
Se necesita descargar 748 kB de archivos.
Se utilizarán 2 519 kB de espacio de disco adicional después de esta operación.
Des:1 http://apt.pop-os.org/ubuntu jammy/main amd64 bison amd64 2:3.8.2+dfsg-1build1 [748 kB]
Descargados 748 kB en 0s (1 610 kB/s)
Seleccionando el paquete bison previamente no seleccionado.
(Leyendo la base de datos ... 514673 ficheros o directorios instalados actualmente.)
Preparando para desempaquetar .../bison_2%3a3.8.2+dfsg-1build1_amd64.deb ...
Desempaquetando bison (2:3.8.2+dfsg-1build1) ...
Configurando bison (2:3.8.2+dfsg-1build1) ...
update-alternatives: utilizando /usr/bin/bison.yacc para proveer /usr/bin/yacc (yacc) en modo automático
Procesando disparadores para man-db (2.10.2-1) ...
root@pop-os:/home/agoca#
```

- 3) Ahora procedemos a instalar el compilador GCC y algunas herramientas de desarrollo necesarias, para ello se ingresa el comando:

sudo apt-get install build-essential

```
root@pop-os:/home/agoca# sudo apt-get install build-essentials
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
E: No se ha podido localizar el paquete build-essentials
root@pop-os:/home/agoca# sudo apt-get install build-essential
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
build-essential ya está en su versión más reciente (12.9ubuntu3).
Fijado build-essential como instalado manualmente.
0 actualizados, 0 nuevos se instalarán, 0 para eliminar y 0 no actualizados.
root@pop-os:/home/agoca#
```

En algunos casos dichas librerías ya vendrán instaladas por defecto o en otros casos y habrán sido instaladas previamente como fue el caso.

Para comprobar que se tiene instalado dicho compilador se puede utilizar el siguiente comando: **gcc --version** si se tiene una salida como la siguiente es que se instaló con éxito, de lo contrario no estará instalado GCC.



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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



```
root@pop-os:/home/agoca# gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

root@pop-os:/home/agoca#
```

- 4) También deberemos tener instalado ya sea Flex o Lex ya que BISON toma entrada de una salida de Flex/Lex, en esta practica no se detallara demasiado sobre esto ya que la practica anterior se centro en esto pero igual se hará mención del archivo que se creara para dicho. Para instalar Flex utilizamos el comando: **sudo apt-get install flex**

```
root@pop-os:/home/agoca# sudo apt-get install flex
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
Se instalarán los siguientes paquetes adicionales:
  libfl-dev libfl2
Paquetes sugeridos:
  flex-doc
Se instalarán los siguientes paquetes NUEVOS:
  flex libfl-dev libfl2
0 actualizados, 3 nuevos se instalarán, 0 para eliminar y 6 no actualizados.
Se necesita descargar 324 kB de archivos.
Se utilizarán 1 148 kB de espacio de disco adicional después de esta operación.
¿Desea continuar? [S/n] s
```

Una vez instalado lo anterior ya estaremos listos para utilizar Bison en su totalidad, así que a continuación se presentaran un ejemplo o caso de uso desarrollado (NOTA: Para simplificar, se utilizara el mismo archivo de Flex para ambos casos para centrarnos mas en explicar la estructura de BISON).

Para mayor comodidad y orden se iniciara creando una nueva carpeta para almacenar lo realizado en las practicas, dicha carpeta yo le daré el nombre de `prac_bisonte`. Y dentro de dicha carpeta para mejor estructura, crearemos dos carpetas adicionales, una llamada “src” que contendrá todo el código fuente y otra llamada “analizador” que contendrá todo el código relacionado a Bison y Flex.



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
[root@pop-os ~]# cd /home/agoca/prac_bisonte/  
[root@pop-os ~]# mkdir src  
[root@pop-os ~]# mkdir analizador  
[root@pop-os ~]# ls  
analizador src
```

Como se menciono anteriormente, se utilizara en este caso el editor de código Sublime Text (aunque se puede utilizar cualquier otro).

Lo primero que realizaremos un controlador mismo que nos será muy útil para el intercambio de datos entre lo que vendría siendo el analizador léxico (FLEX) y el sintáctico (BISON) es importante decir que aunque dicho no es necesario si nos ayudara posteriormente en automatizar el proceso.

Para eso en el directorio de analizador vamos a crear dos archivo uno con nombre “driver.h” y otro con el nombre “driver.cc”

El código de driver.h es el siguiente:

```
#include <string>  
#include "parser.tab.hh"  
  
#define YY_DECL \  
 yy::Parser::symbol_type yylex (Driver& driver)  
YY_DECL;  
  
class Driver {  
public:  
    void runScanner();  
    void closeFile();  
    void parse(const std::string& archivo);  
    std::string file;  
};
```

PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II

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

[Handwritten signatures]

```
driver.h      x  lexical      x  parseryy      x  drivercc
1 #include <string>
2 #include "parser.tab.hh"
3
4 #define YY_DECL \
5     yy::Parser::symbol_type yylex (Driver& driver)
6 YY_DECL;
7
8 class Driver {
9 public:
10     void runScanner();
11     void closeFile();
12     void parse(const std::string& archivo);
13     std::string file;
14 };
15
```

En el anterior código se comienza con la importación de la biblioteca String necesaria para el manejo de cadenas, después se incluye el archivo ”parser.tab.hh” que se revisara mas adelante pero si recordamos este corresponde al archivo que generara bison a partir de una gramática que le suministremos.

Luego corresponde la declaración de la función de escaneo es decir **YY_DECL**. Y después vienen las funciones y atributos necesarios para la clase **Driver**.

En esta se incluye la función ”**runScanner**” que será la encargada de iniciar el análisis, la función ”**closeFile**” que es la encargada de finalizar el análisis. También se incluye la función ”**parse**” que recibirá como parámetro la dirección del archivo de entrada y será la encargada de realizar el análisis sintáctico. Y por ultimo el atributo ”**file**” se usa para almacenar la ruta del archivo de salida.

Ahora respecto al archivo ”**driver.cc**” se tiene el siguiente código:

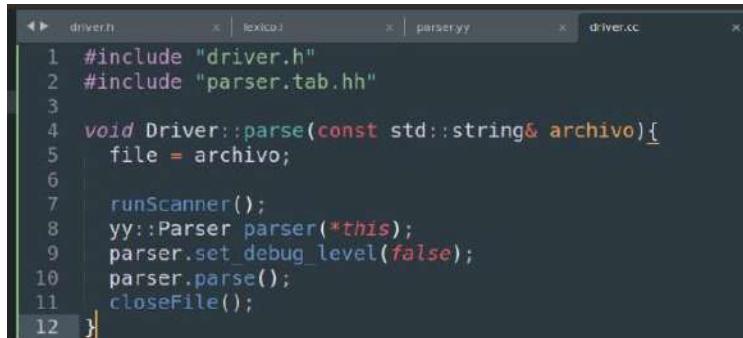
```
#include "driver.h"
#include "parser.tab.hh"

void Driver::parse(const std::string& archivo){
    file = archivo;

    runScanner();
    yy::Parser parser(*this);
    parser.set_debug_level(false);
    parser.parse();
    closeFile();
}
```

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

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



```
driver.h      x | léxico.i      x | parseryy      x | driver.cc
1 #include "driver.h"
2 #include "parser.tab.hh"
3
4 void Driver::parse(const std::string& archivo){
5     file = archivo;
6
7     runScanner();
8     yy::Parser parser(*this);
9     parser.set_debug_level(false);
10    parser.parse();
11    closeFile();
12 }
```

En este primero se incluye el “**driver.h**” que si recordamos es el archivo anteriormente explicado, a continuación de nuevo se incluye el archivo “**parser.tab.hh**”, posteriormente vienen las rutinas que se ejecutaran en la función ”**parse**” que defino en el archivo anterior. Dichas tareas son:

- Se guarda la ruta del archivo en el atributo “**file**”
- **runScanner()** iniciara el análisis
- Se crea un archivo **Parser** que será útil más adelante
- Se usa el método “**set_debug_level(false)**” que indica que no se imprimirá en consola el proceso del analizador.
- Se llama al método **parse** que realizara el análisis sintáctico.
- Y con **closeFile()** se finalizara el proceso de análisis.

A continuación necesitaremos crear un archivo “**léxico.l**” que como su nombre indica contendrá todo el código referente a las instrucciones de Flex.

Dicho archivo también irá dentro de la carpeta de “**análizador**” a continuación se muestra el contenido de dicho archivo aunque no se explicara mas que algunos detalles notables ya que no es la finalidad de la práctica en curso.

```
%{
#include <stdio.h>
#include <string>
#include "driver.h"
#include "parser.tab.hh"
}

%option noyywrap
%option outfile="scanner.cc"
```



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

[Handwritten signatures]

```
DIGIT [0-9]
NUM  {DIGIT}+("."+{DIGIT}+)?  

%%  

{NUM}    {return yy::Parser::make_NUM(strtol (yytext, NULL, 10));}
"evaluar" {return yy::Parser::make_EVALUAR();}
"+"      {return yy::Parser::make_MAS();}
"-"      {return yy::Parser::make_MENOS();}
"*"      {return yy::Parser::make_POR();}
"/"      {return yy::Parser::make_DIV();}
 "("      {return yy::Parser::make_PARIZQ();}
 ")"      {return yy::Parser::make_PARDER();}
 ";"      {return yy::Parser::make_PTOCOMA();}  

  
[[:blank:]]  {}
.           {printf("Caracter no reconocido: %s\n",yytext);}
<<EOF>>      {return yy::Parser::make_FIN();}
%%  

void Driver::runScanner(){
    yy_flex_debug = false;
    yyin = fopen (file.c_str (), "r");
    if(yyin == NULL){
        printf("No se encontro el archivo de entrada");
        exit(1);
    }
}

void Driver::closeFile(){
    fclose(yyin);
}
```

PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II

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

[Handwritten signatures]

```
1 // driver.h
2 // lexico.l
3 // parser.y
4 // driver.cc
5
6
7 %option noyywrap
8 %option outfile="scanner.cc"
9
10 DIGIT [0-9]
11 NUM {DIGIT}+("."{DIGIT}+)??
12 %%
13
14 {NUM} {return yy::Parser::make_NUM(strtol(yytext, NULL, 10));}
15 "evaluar" {return yy::Parser::make_EVALUAR();}
16 "+"
17 "-"
18 "*"
19 "/"
20 "("
21 ")"
22 ","
23 ";"
24 "[[:blank:]]" {}
25 "."
26 <<EOF>> {printf("Carácter no reconocido: %s\n", yytext);}
27 %%
28 void Driver::runScanner(){
29     yy_flex_debug = false;
30     yyin = fopen(file.c_str(), "r");
31     if(yyin == NULL){
32         printf("No se encontró el archivo de entrada");
33         exit(1);
34     }
35     void Driver::closeFile(){
36         fclose(yyin);
37 }
```

Algunas notaciones importantes es que el archivo que generara Flex se llamará “scanner.cc” y se incluyen dos expresiones regulares que identificaran enteros y decimales (DIGIT y NUM). Por ultimo se incluyen las reglas que van a ser utilizadas en el analizador sintáctico (BISON), mismas que tendrán la estructura de **make_** seguido del nombre que tendrá dicho token.

Contaremos con los siguientes Tokens:
NUM,EVALUAR,MAS,MENOS,POR,DIV,PARIZQ,PARDE,PTOCOMA,FIN.

Por ultimo se incluye la ejecución del escaneo que buscara el archivo de entrada.

Finalmente pasaremos a revisar el archivo que mas nos compete es decir “parser.y” que recordemos es el archivo que contendrá todas las instrucciones que BISON necesitará llevar a cabo.

El código de dicho archivo es el siguiente:



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
%skeleton "lalr1.cc" /* -*- C++ -*- */\n\n%defines\n#define api.parser.class {Parser}\n#define api.token.constructor\n#define api.value.type variant\n\n#define parse.trace\n#define parse.error verbose\n%param { Driver& driver }\n\n\n%code requires\n{\n    class Driver;\n}\n\n%\n\n#include <string>\n#include <stdio.h>\n#include "driver.h"\n#include <iostream>\n%\n\n\n***** TERMINALES *****\n%token MAS "+" MENOS "-" POR "*" DIV "/" PARIZQ "(" PARDER ")" PTOCOMA ";" EVALUAR "EVALUAR"\n%token <float> NUM "NUM"\n%token FIN 0 "eof"\n\n%left MAS MENOS\n%left POR DIV\n%precedence NEG
```



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
***** NO TERMINALES *****/  
%type <float> Expr  
%start Inicio;  
  
%%  
  
Inicio  
: Lista ;  
  
Lista  
: Lista "EVALUAR" "(" Expr ")" ";"  
{  
    std::cout<< "El valor de la Expresion es: " << $4 << "\n";  
}  
| "EVALUAR" "(" Expr ")" ";"{  
    std::cout<< "El valor de la Expesion es: " << $3 << "\n";  
}  
;  
  
Expr  
: Expr "+" Expr {  
    $$ = $1 + $3;  
}  
| Expr "-" Expr {  
    $$ = $1 - $3;  
}  
| Expr "*" Expr{  
    $$ = $1 * $3;  
}  
| Expr "/" Expr{  
    $$ = $1 / $3;  
}  
| "-" Expr %prec NEG{  
    $$ = -$2;
```



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
}

| "NUM"
{
    $$ = $1;
}

| "(" Expr ")"
{
    $$ = $2;
}

;

%%

void yy::Parser::error( const std::string& error){
    std::cout << error << std::endl;
}
```

Debido a la extensión del código se procederá a explicar en bloques o secciones de código. Así pues revisemos el primero

```
1 %skeleton "lalr1.cc" /* -*- C++ -*- */
2
3 %defines
4 %define api.parser.class {Parser}
5 %define api.token.constructor
6 %define api.value.type variant
7
8 %define parse.trace
9 %define parse.error verbose
10 %param { Driver& driver }
```

Esta parte corresponde a las directivas de BISON. En la primera se indica que el tipo de analizador que queremos que se ejecute es un analizador LALR(1), posteriormente las siguientes le indican a BISON el nombre de la clase a generar (es decir constructor) y las siguientes de “parse.trace” y “parse.error” se colocan para el manejo de errores sintácticos. Y en driver se indica a bison que la clase que va a generar recibirá un parámetro de tipo driver.

Ahora pasemos a la siguiente sección:



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

```
%code requires
{
    class Driver;
}

%{

#include <string>
#include <stdio.h>
#include "driver.h"
#include <iostream>
%}
```

En esa sección se incluyen las bibliotecas, librerías y cabeceras que serán necesitadas para la ejecución del analizador sintáctico (bison). La clase Driver es necesario para agregar las funciones establecidas en dicha clase.

Ahora viene la sección de los símbolos terminales

```
/****** TERMINALES *****/
%token MAS "+" MENOS "-" POR "*" DIV "/" PARIZQ "(" PARDER ")" PTOCOMA ";" EVALUAR "EVALUAR"
%token <float> NUM "NUM"
%token FIN 0 "eof"

%left MAS MENOS
%left POR DIV
%precedence NEG
```

Aquí mediante los tokens de parte de Flex procedemos a definir el símbolo que identificara dicho token, vemos que a algunos se les establece el tipo esto no es tan necesario ya que en muchos casos únicamente se requiere colocar su símbolo asociado, pero en el caso de “NUM” si se le coloca su tipo ya que mediante este vamos a realizar operaciones por lo cual su tipo se vuelve importante y necesario.

Ya en la parte debajo nos encontramos la definición del nombre de precedencia de operadores ya que sin esta la gramática puede tornarse ambigua para ello se define desde el mas bajo al más alto, esto siguiendo el principio del orden de los operadores que utilizamos en matemáticas. Es decir el que tiene menos importancia o peso es el más (+) y menos (-), seguido por la multiplicación (*) y la división (/) y por ultimo incluimos el símbolo negativo asociado a expresiones negativas(es decir el que usamos para negar una operación o en otras palabras multiplicar por -1)

A continuación tenemos las producciones o también conocidas como reglas de escritura de la gramática.

PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II

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

[Handwritten signatures]

```
36 /****** NO TERMINALES *****/
37 %type <float> Expr
38 %start Inicio;
39
40 %%
41
42     Inicio
43         : Lista ;
44
45
46     Lista
47         : Lista "EVALUAR" "(" Expr ")" ";"
48         {
49             std::cout<< "El valor de la Expresión es: " << $4 << "\n";
50         }
51     | "EVALUAR" "(" Expr ")" ";"{
52         std::cout<< "El valor de la Expresión es: " << $3 << "\n";
53     }
54 ;
```

```
56     Expr
57         : Expr "+" Expr {
58             $$ = $1 + $3;
59         }
60         | Expr "-" Expr {
61             $$ = $1 - $3;
62         }
63         | Expr "*" Expr{
64             $$ = $1 * $3;
65         }
66         | Expr "/" Expr{
67             $$ = $1 / $3;
68         }
69         | "-" Expr %prec NEG{
70             $$ = -$2;
71         }
72         | "NUM"
73         {
74             $$ = $1;
75         }
76         | "(" Expr ")"
77         {
78             $$ = $2;
79         }
80 ;
81 %%
82 %%
```

Si se requiere se puede definir el tipo de las no terminales a emplear, en este caso únicamente se asigna un tipo de dato **float** al no terminal **Expr**.

Posteriormente se indica cual será el símbolo inicial (es decir Inicio) y se procede a escribir las producciones correspondientes para ello se utilizan las llaves “{}”, como se había revisado anteriormente aquí se puede escribir código en C. Y ya que bison trabaja con analizadores ascendentes, se permite el sintetizar atributos para realizar esto se hace uso del identificador de doble dólar “\$\$”

Posteriormente las producciones Inicio y Lista definen como se construyen dichas a partir de otras estructuras y tokens. En esta ocasión **Inicio** se construye a partir de una **Lista** y a

PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II

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

[Handwritten signatures]

su vez **Lista** se puede construir por medio de otra **Lista** seguida por la secuencia ya sea “**EVALUAR**” “(“ **Expr** ”)” “;” o “**EVALUAR**” “(“ **EXPR** ”)” “;”.

EXPR está definido con múltiples reglas para poder representar las operaciones matemáticas básicas (+,-,*,/) y cada una de estas acciones tienen una acción asociada que realiza el cálculo que corresponde.

“-” **Expr %prec NEG** es una regla que permitirá negar una expresión. **%prec NEG** establece la presedencia de dicha reglas para resolver o evitar conflictos de análisis.

“**NUM**” y “(“ **Expr** ”)” son reglas que permiten numero individuales y expresiones entre paréntesis como expresiones validas.

En todas estas anteriores reglas, **\$\$** se usa para referirse al valor semántico de la estructura que se está construyendo, mientras que **\$n** se usa para referirse al valor semántico del enésimo componente de la estructura.

Ya como ultimo proceso necesitaremos dar los archivos a Flex para que genere un analizador léxico en base al código fuente del archivo léxico y también necesitaremos indicara a bison que genere los archivos de compilación para el analizador sintactico en base al archivo Parser.

Para ello necesitaremos ingresar los siguientes dos comandos:

Flex lexico.l

bison parser.yy

Dichos comandos anteriores darán de salida los siguientes archivos:

- **parser.tab.cc**
- **parser.tab.hh**
- **scanner.cc**
- **stack.hh**

```
src
  analizador
    / driver.cc
    / driver.h
    / lexico.l
    / parser.tab.cc
    / parser.tab.hh
    / parser.yy
    / scanner.cc
    / stack.hh

2 [agocaa@pop-os] [-/pra_bisonte/src]
3   $ cd analizador/
4   [agocaa@pop-os] [-/pra_bisonte/src/analizador]
5   $ flex lexico.l
6   [agocaa@pop-os] [-/pra_bisonte/src/analizador]
7   $ bison parser.yy
8   [agocaa@pop-os] [-/pra_bisonte/src/analizador]
9   $
```

Ahora lo único que necesitamos es un archivo de entrada que se llamara “**entrada.txt**” (este lo colocamos dentro de src pero fuera de la carpeta de analizador) que como su



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

[Handwritten signatures]

nombre dice tendrá el archivo de entrada que reconocera nuestros analizadores.
El contenido de dicho es el siguiente:

```
evaluar(1+1);
evaluar(1+1*2);
evaluar(-(1+1*6/3-5+7));
evaluar(-(1+1*6/3-5+1*-2));
evaluar(-(1+1));
```

Y por ultimo necesitamos crear un archivo main.cc que tomara el archivo a analizar.
Dentro de este colocamos lo siguiente:

```
#include <stdio.h>
#include "./analizador/driver.h"

int main() {
    Driver driver;
    driver.parse("entrada.txt");
    return 0;
}
```

Para finalizar y ejecutar nuestra aplicación primero la necesitamos compilar con el compilador de c++ con el comando: **c++ main.cc ./analizador/*.cc**

Esto anterior nos generara un archivo a.out que podremos ejecutar en consola con el resultado con el comando **./a.out**

```
[agoca@pop-os] -[~/pra_bisonte/src]
└─ $ ./a.out
    El valor de la Expresion es: 2
    El valor de la Expresion es: 3
    El valor de la Expresion es: -5
    El valor de la Expresion es: 4
    El valor de la Expresion es: -2
```



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA

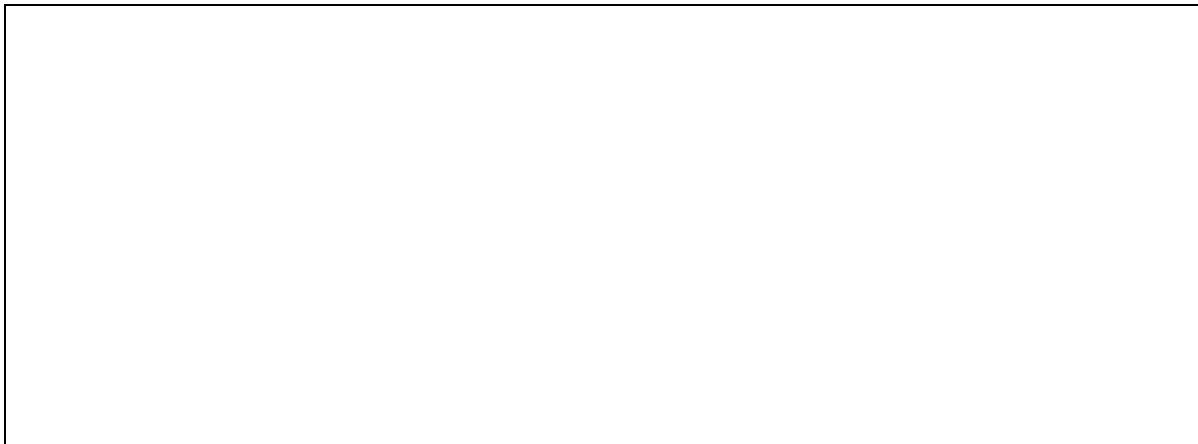


PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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





BÍTACORA DE INCIDENCIAS		
FECHA	PROBLEMA ENCONTRADO	SOLUCIÓN
8/10/23	Falta de información relevante respecto a Bison.	Realizar la búsqueda en inglés o consultando la documentación oficial de GNU
9/10/23	Problemas de escritura de carpetas en Linux	Al realizar las carpetas en root, el usuario convencional no podía acceder a estas e igual los editores de texto, se rehizo la carpeta como usuario normal para corregir esto.
9/10/23	Problemas de conectividad de un compañero para realizar el video	Enviarle una recarga de 20 pesos de internet ilimitado para que pudiera atender la sesión.

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

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



7

CONCLUSIÓN

Mediante la práctica anterior pudimos abarcar de forma práctica el concepto del analizador sintáctico de Bison lo cual resultó bastante útil ya que estos conceptos pueden parecer un tanto extraños o difíciles de captar sin la parte práctica para fortalecer la teoría. Al mismo tiempo también se retomaron conceptos de FLEX para poder llevar a cabo dicha práctica pero gracias a la actividad realizada anteriormente dicho procedimiento no supuso un gran problema. Mediante el desarrollo de el ejemplo práctico que se llevó a cabo sirvió tanto para reforzar los conocimientos previos de analizadores léxico como también abordar el tema de los sintácticos a detalle.

Se pudo comprender el código que requiere Bison como lo vendrían siendo la declaración de expresiones o patrones que de hecho pudimos darnos cuenta de que era muy similar a la de Flex/Lex lo cual tiene sentido considerando que ambas herramientas van de la mano es su trabajo.

También se pudo comprobar que su relación con YACC es muy estrecha ya que ambos son prácticamente hermanos gemelos pero en cuestión y a grandes rasgos Bison es de código abierto, mejora el manejo de errores y es más personalizable.

Para finalizar gracias a esta práctica se amarraron cabos sueltos de la teoría de los analizadores sintácticos al unir esta con la parte teórica de esta práctica.

8

REFERENCIAS

Donnelly, C. D., & Stallman, R. S. (2021). Bison The Yacc-compatible Parser Generator (3.8.1). Free Software Foundation, Inc.

<https://www.gnu.org/software/bison/manual/bison.pdf>

Declarations (Bison 3.8.1). (s. f.).

https://www.gnu.org/software/bison/manual/html_node/Declarations.html

Grupo. (2017, 29 agosto). Compilador Con Bison y Flex. BLOG DE COMPILEDORES.

<https://analizadorsemanticolab1.blogspot.com/2017/08/complidor-con-bison-y-flex.html>

Manrique, M. M. R. (s. f.). HERRAMIENTAS YACC Y BISON. Biblioteca de la Universidad Nacional del Santa.



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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



http://biblioteca.uns.edu.pe/saladocentes/archivoz/publicacionez/sesion_ii_3u_herramienta_yacc_y_bison.pdf

Mi primer proyecto utilizando Yacc y Lex. (2020, 1 octubre). Erick Navarro.

<https://www.ericknavarro.io/2020/10/01/27-Mi-primer-proyecto-utilizando-Yacc-y-Lex/>

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

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

[Handwritten signatures]

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

PRACTICA NO.	NOMBRE DE LA PRACTICA	FECHA DE ENTREGA
1	COMPRENDIENDO ASML EUV Y EL POR QUÉ ASML SUPONE UN MONOPOLIO INMINENTE EN LA INDUSTRIA DE CHIPS	9/10/23

1	INTRODUCCION
ASML es la única compañía que fabrica máquinas de litografía del tamaño de un autobús que son capaces de producir los chips más avanzados del mundo logrando la impresión de transistores más pequeños que un cabello humano. En este documento se van a abarcar el proceso de fabricación de un chip y que impresoras intervienen en el proceso, con el fin de comprender en parte la importancia que tiene ASML en el ámbito de los chips.	

2	OBJETIVO
Mediante la visualización del video y los apuntes de este mismo se espera el comprender e informarnos acerca de la nueva máquina litográfica de la empresa ASML que tiene el nombre de ASML EUV y entender por qué dicha tecnología supone una gran innovación tecnológica que abre la puerta a chips más rápidos y derivado de lo anterior darnos cuenta porque esto supone un monopolio sin precedentes para ASML ya que ninguna otra empresa cuenta con tal grado de tecnología.	

3	FUNDAMENTO
ASML es una empresa neerlandesa que se dedica a la fabricación de máquinas para la producción de circuitos integrados. Es el mayor proveedor del mundo de sistemas de fotolitografía para la industria de los semiconductores. Las máquinas de fotolitografía son utilizadas para producir chips de computadoras y otros dispositivos electrónicos avanzados. En estas máquinas, los patrones son proyectados ópticamente sobre una oblea de silicio que es cubierta con una película de material sensible a la luz, y el procedimiento se repite muchas veces en una misma oblea. Un semiconductor es un material que, dependiendo de varios factores, permite el paso de corriente eléctrica o no. Son especialmente utilizados en la fabricación de componentes electrónicos que se encuentran en la mayoría de los dispositivos tecnológicos.	



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

Un chip es una placa muy pequeña de material semiconductor que incorpora un circuito electrónico. Está compuesto por millones de componentes electrónicos microscópicos llamados transistores que se encargan de transmitir señales de datos.

4

MATERIALES NECESARIOS

Hardware:

- CPU: Intel i3 8va generación 2.4 GHz
- RAM: 8GB
- SSD: 128 GB

Software:

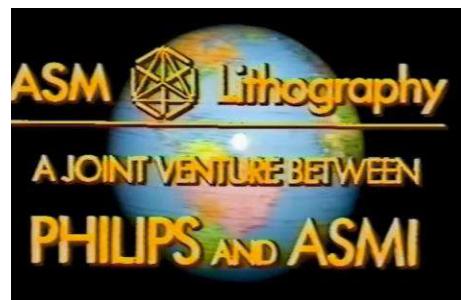
- SO: Windows 10 o alguna distribución de Linux.
- Algún navegador de internet

5

DESARROLLO

Éxito

Para lograr el éxito se necesita superar obstáculos, en este caso uno de los más grandes fue el de “La ley de Moore” la cual consiste en el hecho de que cada dos año se duplica el número de transistores de un chip, de esta manera dando comienzo con la unión de científicos que comparten una visión. En 1984 se unen Phillips se unen para dar pie a la fundación de la compañía Advanced Semiconductor Materials Lithography o ASML.



En 1985 lanzan la primera máquina de la época que usaba rayo de luz de manera muy precisa para imprimir diseños en silicio de esta manera fabricando chips, esta tecnología se conoce como fotolitografía. En 1986 el mercado era dominado por Nico ni Canon mientras que a ASML se le

**PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II****AUTORES:**

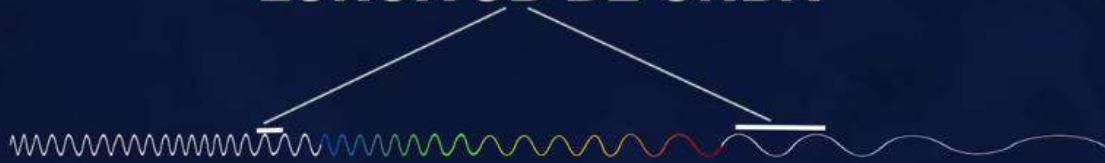
Alma Gabriela Ponce Morales
Anthony Gómez Cabañas
Cristhian Alberto Ortega Hernández
Miguel Ángel Ruiz López

terminaba el dinero, pero decidieron invertir esos fondos restantes en producir una maquina llamada Twin scan, esta máquina combinaba el proceso de medición e impresión de los chips aumentando la velocidad de fabricación. El éxito de ASML no se debe solo a los avances tecnológicos si no en las decisiones de inversión tomados en momentos de crisis cuando nadie más invertía.

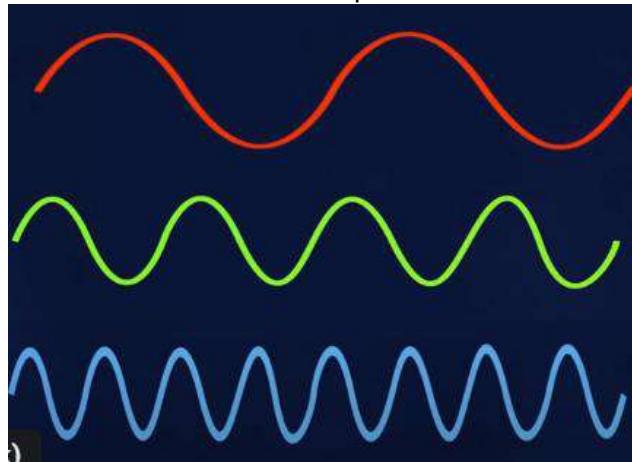
3. LA LUZ

Para hacer chips, usan luces de mayor frecuencia para darle menor longitud de onda y así dar mayor gama de colores a ASML. Cuanta mayor frecuencia, menor es la longitud de onda:

LONGITUD DE ONDA



Además, la frecuencia también determina el color que transmiten las ondas:



ASML es capaz de crear una luz de una longitud de onda de 13.6 nm llamada Extreme Ultraviolet (EUV), la cual es una longitud de onda catorce veces menor que la de la máquina de twinscan, y por ello, puede imprimir chips con transistores que son 1000 veces más pequeños que un pelo humano.

La idea de implementar una luz de frecuencia muy baja era una idea costosa y arriesgada cuando se quisiera materializar en un proyecto, pero ASML logró encontrar una solución al hacer que sus clientes más importantes compraran una parte de las acciones de ASML, pudiendo así, adentrarse en el desarrollo de una de las máquinas más complejas y costosas jamás hechas



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

llamada EUV. El desarrollo y éxito de esta máquina pondrán a ASML como uno de los fabricantes de chips más rápidos actuales al grado de llegar a un monopolio.

4. LA MÁQUINA EUV

Para entender el funcionamiento de la máquina EUV, es necesario entender conceptos de física y computación. Los chips están en todas partes, en casi cualquier dispositivo electrónico que se pueda tener, los smartphones son una prueba clara de ello. Estos chips están distribuidos por pasadizos y secciones diseñados microscópicamente para enviar y recibir información con tan solo 1s y os:

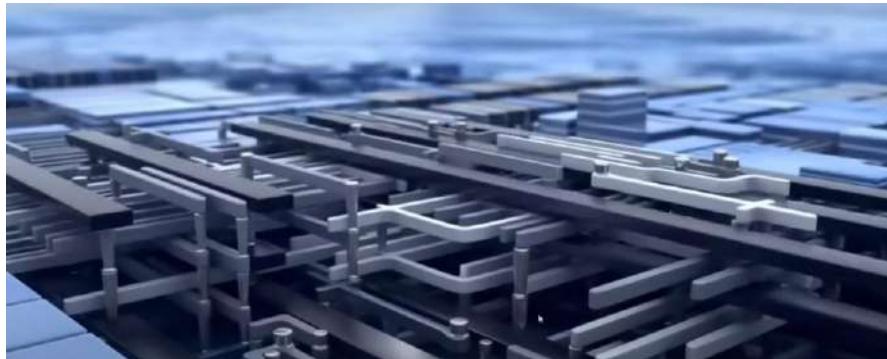


Figura. Pasadizos de un Chip

Cada letra, número, símbolo y carácter del alfabeto latino, arábigo, devanagari (India) o cualquier otro alfabeto digitalizado tiene correspondiente una cadena equivalente en binario. Incluso cada pixel de la pantalla del smartphone es de un color, y ese color tiene su equivalente en binario también. Para mostrar todo el contenido en la pantalla, el chip (microprocesador) del teléfono debe realizar millones de operaciones en binario por segundo por lo que, mientras más operaciones por segundo haga, mayor será la rapidez del dispositivo.

Estos 1s y os que componen a cada pixel, cada color y a cada símbolo son creados a través de conceptos de electrónica y semiconductores.

Si se observa el ejemplo de un cable de cobre, se estima que los electrones de sus átomos se mueven de forma aleatoria:



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

[Handwritten signatures]

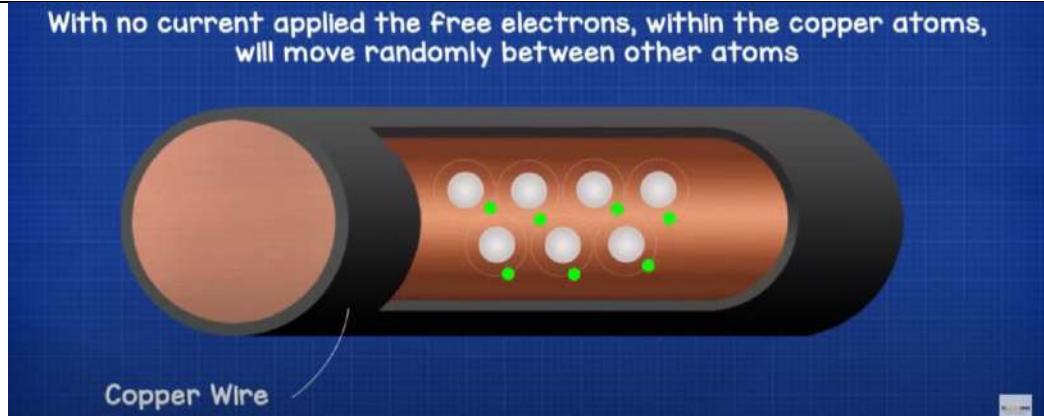


Figura. Átomos moviéndose aleatoriamente

Pero cuando sus 5 electrones se mueven en una misma dirección, se dice que se genera electricidad, la cual es básicamente el flujo de 5 electrones de un punto a otro.

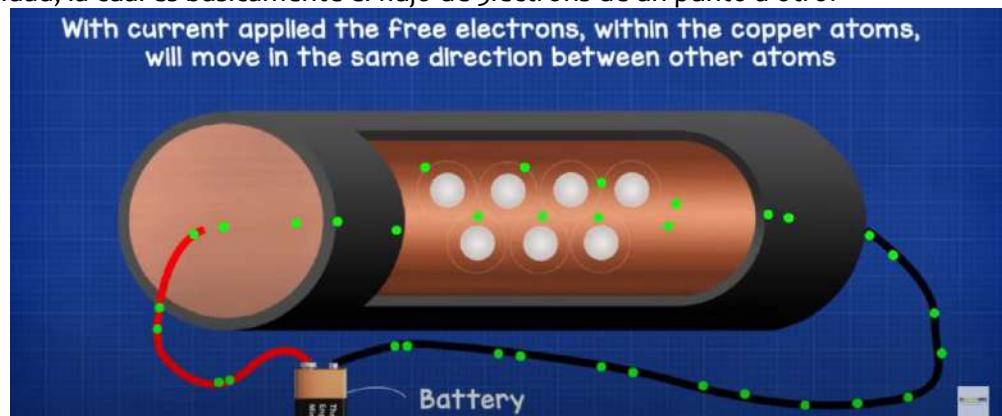


Figura. Átomos moviéndose en la misma dirección

Sin embargo, la electricidad generada por un flujo continuo de electrones no puede ser transformada en 1s y 0s, para ello, se utilizan materiales, llamados semiconductores, que puedan a la vez actuar como aislantes (como el silicio) que puede interpretarse como un 1 en caso de conducir corriente y en 0 en caso de aislarla.



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

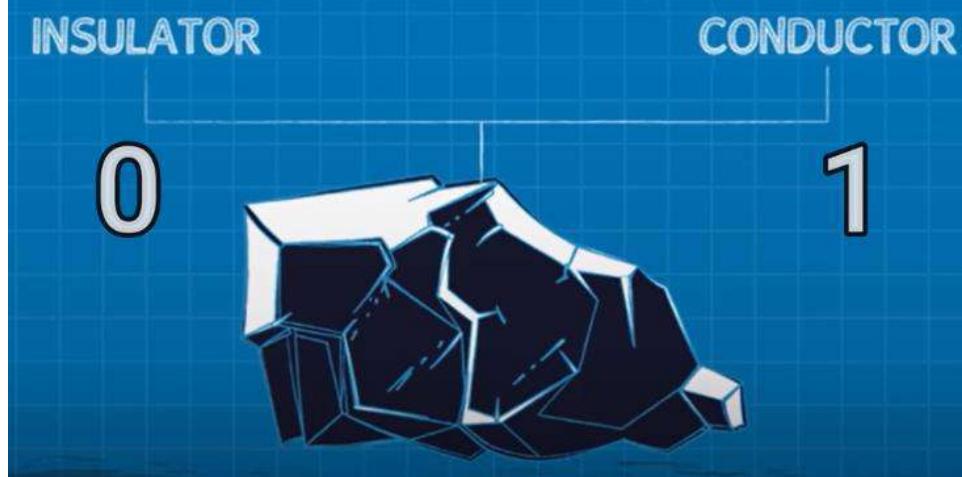


Figura. Materiales Semiconductores

Los componentes electrónicos llamados transistores fueron creados para controlar la generación de estos 1s y 0s de forma controlada y no aleatoria. Su diseño es similar al de un chip, en el sentido que está dividido en series de caminos y pasadizos por donde circula la electricidad para que puedan o no darle paso dependiendo de las instrucciones de control del chip o microprocesador.

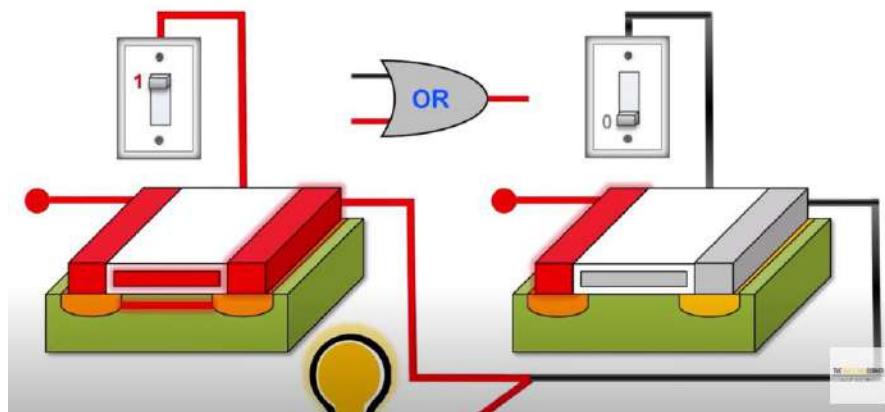


Figura. Control izquierdo encendido

PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II

AUTORES:

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

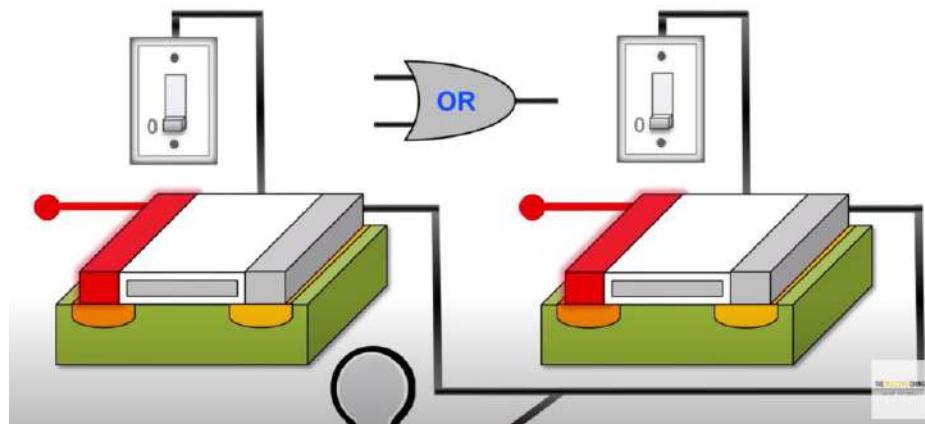


Figura. Control izquierdo y derecho apagados

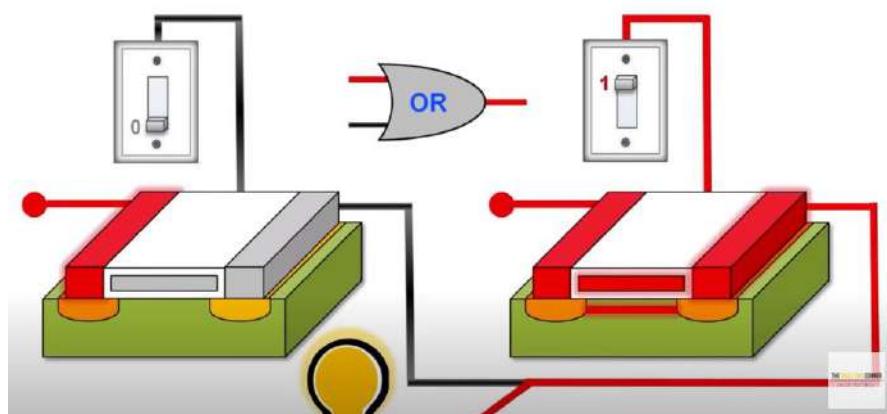
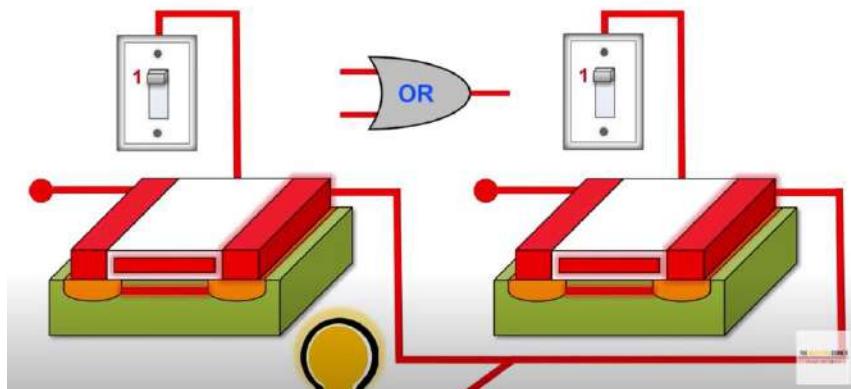


Figura. Control derecho encendido





TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

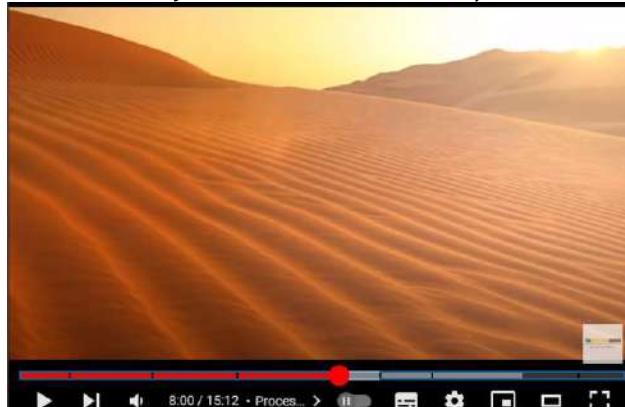
Figura. Control izquierdo y apagado encendidos

De esta forma, el transistor puede servir como un interruptor, controlando el encendido y apagado de dispositivos a través de 1s y os generados de manera controlada.

Cuantos más transistores o interruptores tengan un chip, más 1s y os podrá procesar y por lo tanto, más rápido será. Si se desean hacer chips más potentes y rápidos, se deben crear transistores cada vez más pequeños para poder hacer que quepan más. Las máquinas de ASML son capaces de diseñar estos caminos de los transistores.

Proceso de fabricación de un chip

El primer paso consiste en la recolección de arena debido a que dicha contiene altos niveles de silicio, este elemento se usa debido a que es el semiconductor más abundante en la tierra (a diferencia de otros como el galio) y a su facilidad de conversión a semiconductor (es decir bajo ciertos casos actúa como conductor y en otros como aislante).



Los pasos que se siguen son los siguientes: Primero se purifica la arena para extraer el silicio y a partir de este realizar cilindros alargados de silicio.



TECNOLÓGICO
NACIONAL DE MÉXICO

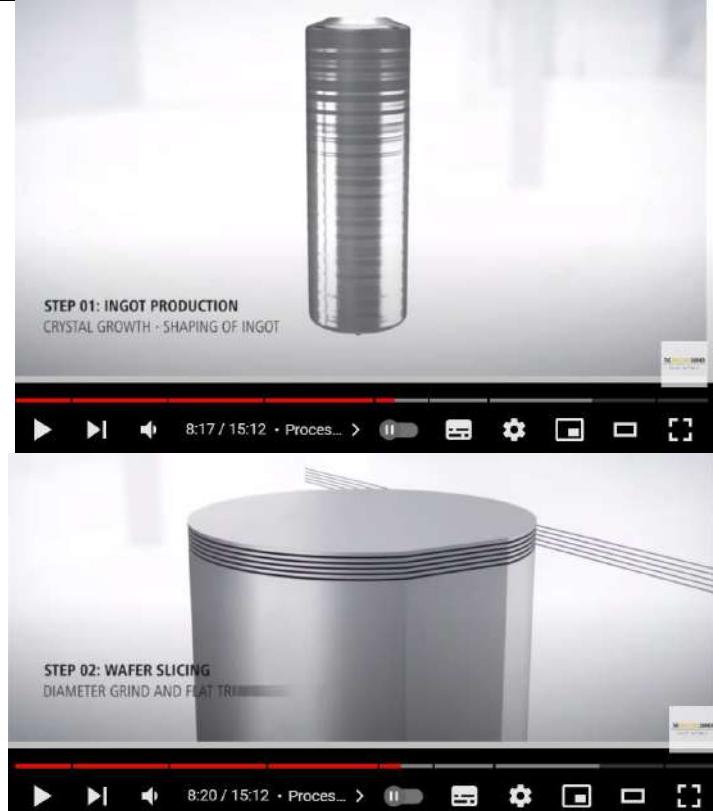
INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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



Segundo se parte dicho cilindro se corta a ancho para crear lo que se denomina un wafer (o traducido como oblea debido a su forma) pudiendo ser de diferente tamaño según se requiera. Tercero se adelgaza y se pule la oblea. Sobre esta oblea es sobre la cual se van a ir posicionando los elementos capa a capa como si fuera un sándwich. Cuarto se aplica una capa fotorresistente para que cuando se le aplique iluminación esta se endurezca.





TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

[Handwritten signatures]



Quinto se expone la oblea a iluminación ultravioleta, pero dicha iluminación se aplica de una forma muy específica es decir a través de un lente óptico con una máscara (diseño con patrones) específica que va serigrafiando unos patrones en concreto sobre la oblea, esta se conoce como fotomáscara y el proceso es conocido como ASML EUV, si contamos con un microscopio podemos ver que después de pasar dicha luz sobre la oblea se puede apreciar trazos o caminos en la oblea.

Funcionamiento de la luz EUV

Para que la maquina ASML EUV produzca el haz de luz primero se comienza lanzando una cantidad pequeñísima pero abundante de gotas de estaño, aproximadamente cincuenta mil por segundo. Después se emite un haz de luz que incide en cada una de las gotas (a veces incide hasta dos veces), este proceso genera unas explosiones pequeñísimas que crea el plasma que posteriormente emitirá la luz ultravioleta (en concreto esta luz es de 13.5 nanómetros es decir 13.5×10^{-9}) esta luz viajara a través de los lentes o espejos y a través de la máscara para posteriormente impactarse en la oblea imprimiendo sobre esta los trazos o caminos.





TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

[Handwritten signatures]



A partir de esto siguen otros procesos de aplicación de materiales algunas siendo necesarias para eliminar material de áreas aun no expuestas a la luz, otras capas ionizan al silicio, en otra se elimina la fotorresistencia y se procede a aplicar una capa de metal por donde circulará la electricidad se procede a pulir siendo este el paso final del ciclo, ¿Por qué decimos que un ciclo? Pues porque este proceso se repetirá muchas veces más eh de ahí el nombre de estructura de fabricación 3D de chips esto para formar cada trazo necesario de la oblea.



Este proceso es revolucionario y le otorga un monopolio absoluto a ASML debido a que cuenta con la mejor tecnología que será base para crear chips cada vez más rápidos.

El sector de semiconductores

Existen muchas compañías que participan en el momento de fabricar un chip:

- Las compañías de software, como Cadence o Synopsys, participan en el diseño del chip.
- Hay otras compañías que se encargan principalmente del diseño del chip, como Nvidia, AMD o Broadcom.
- Otras compañías se encargan de la venta de maquinaria, com ASML, HITACHI, entre otras.

Para poder realizar un mejor chip, es necesario que la máquina de ASML pueda fabricar transistores



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

[Handwritten signatures]

más pequeños. Hay compañías que están interesadas en desarrollar una tecnología similar a la que ya tiene ASML, pero esto es muy complicado porque, además de ser una tecnología muy difícil de desarrollar, ASML se encuentra actualizando y mejorando su tecnología constantemente, por lo que sería muy difícil igualarlos. Actualmente, ASML ya cuenta con una nueva máquina que está lista para ser lanzada al mercado en 2024, y tendrá un costo de 350 millones de euros.

Esta nueva máquina se llama High NA, y cuenta con una apertura numérica mayor que permite reducir el tamaño de la impresión de los transistores. Empresas como TSMC ya anunciaron que estarán desarrollando chips de 2 nanómetros para 2025.

La competencia en este sector es muy limitada, ASML es prácticamente la dueña del mercado de chips. ASML tiene contratos exclusivos con sus proveedores, por lo que, si otra compañía quiere entrar a competir, necesitará hablar con otros proveedores que no tengan experiencia.

A ASML se le culpa porque el sector de los semiconductores es muy cíclico, y mucha gente cree que cuando la demanda de los semiconductores disminuya, ASML se encontrará en crisis, pero esto no es así. Todas las industrias entran en recesión porque cuando se expanden se invierte mucho en nuevas fábricas, es por eso por lo que llega un momento en el que la oferta supera la demanda y los precios comienzan a caer y las empresas pierden dinero. En la industria de los semiconductores, estas recesiones van disminuyendo, como se puede observar en la gráfica.

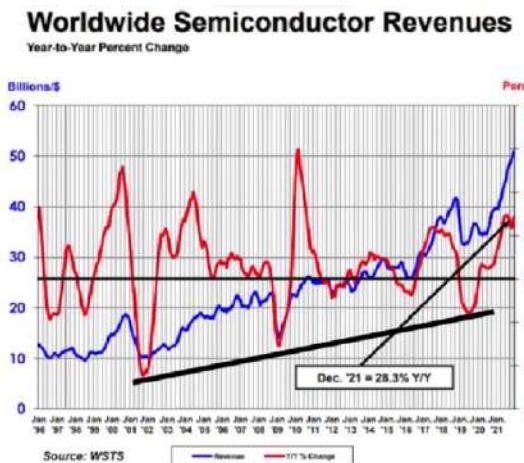


Figura. Recesión del sector de chips

Esto es debido a que la demanda siempre está por delante de la oferta. Gracias a tecnologías como el Cloud Computing, IoT o Inteligencia Artificial, la demanda de los chips crece de manera exponencial pero también la oferta se reduce; cada vez son menos las compañías que se dedican al desarrollo de

**PRÁCTICA DE LABORATORIO - LENGUAJES
Y AUTÓMATAS II****AUTORES:**

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

chips. Hoy en día, sólo hay una compañía que fabrica el 92% de los chips de 5 nanómetros, y es TSMC. El costo de la máquina de ASML es muy alto y es por eso que muy pocas compañías pueden comprarla, por ejemplo, TSMC, Intel y Samsung.

TSMC siempre invierte en la parte baja, por eso no le importa que venga un mal ciclo para los semiconductores.



Figura. Preferencia de inversión de TSMC

Si invierte cuando la demanda está abajo, cuando se recupere la demanda robará cuota de mercado de todas las empresas que no hayan invertido.

Números y proyecciones de ASML

ASML tiene el monopolio de los chips más potente en un sector que crece estructuralmente, es por eso por lo que sus ingresos siguen creciendo a pesar de tener una capitalización bursátil de 230 mil millones. Puede generar retornos del capital del 50% y aun así tiene capacidad de mejorar año con año a medida que venda las máquinas. Además, es menos cíclica que el sector de los semiconductores y esto hace que la visibilidad de sus beneficios sea más alta de lo normal

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

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

6	BÍTACORA DE INCIDENCIAS		
Fecha	Problema encontrado	Solución	
9-10-2023	Fallas en el internet.	Esperar a que vuelva a haber internet.	
9-10-2023	No se actualizaba el documento compartido.	Cerrar y volver a abrir el documento.	
9-10-2023	Una laptop de uno de los miembros del equipo no encendía sin estar conectada a la corriente.	Se conectó a la corriente de luz y ya se pudo trabajar.	

7	CONCLUSIÓN
<p>En el video se presentó información relevante sobre la fabricación de chips y la tecnología que la respalda. Se abordan conceptos clave, como la influencia de las frecuencias de la luz y el papel central de las máquinas EUV de ASML en esta industria. Además, se destaca cómo ASML ha adquirido una posición dominante en el mercado gracias a su tecnología avanzada y una inversión significativa en investigación y desarrollo. El proceso de producción de chips, desde la litografía hasta las etapas de fabricación, se presenta como un procedimiento altamente sofisticado y meticuloso.</p> <p>Por último, este video ofrece una visión detallada de cómo la tecnología está impulsando avances en la industria de los semiconductores, lo que tiene un impacto directo en nuestra vida cotidiana en la era digital actual.</p>	



TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA



PRÁCTICA DE LABORATORIO - LENGUAJES Y AUTÓMATAS II

AUTORES:

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

8

REFERENCIAS

- [1] The Investing Corner. (2023, 18 septiembre). Por qué ASML tiene el monopolio más COLOSAL de la historia [Vídeo]. YouTube.
https://www.youtube.com/watch?v=zAYCfw_syFc
- [2] López, J. C. (2023). ASML ya tiene lista su revolucionaria máquina EUV High-NA. es un giro crucial en la guerra comercial de... Xataka.
<https://www.xataka.com/empresas-y-economia/giro-imprevisto-guerra-tecnologica-eeuu-china-asml-tiene-lista-su-revolucionaria-mquina-euv-high-na>
- [3] Drex. (2023). The brain behind the machine: Transistors in CPU architecture. DRex Electronics. <https://www.icdrex.com/the-brain-behind-the-machine-transistors-in-cpu-architecture/#:~:text=Q%3A%20How%20do%20transistors%20work,binary%20data%20in%20memory%20cells>.