

TECNOLÓGICO NACIONAL
DE MÉXICO EN
CELAYA

Asignatura: LENGUAJES & AUTOMÁTAS II

Docente: ISC. Ricardo González González

2023-10-02 LA-II-A

Actividad 5º Análisis Sintáctico

EQUIPO #1

INTEGRANTES:

- ▷ Arroyo Gómez José ALFREDO (20030029)
- ▷ García Hernández CESAR (20030853)
- ▷ Gasca Palacio JESÚS FERNANDO (20030606)
- ▷ González Mancera CHRISTIAN MANUEL (20030115)

2 = var int X

int var = 2 ✓

{ ¿El para que cosa
de quién? }

Ser sintácticamente correcto
ayuda a evitar errores en
la compilación



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.





- 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.

César García Hernández



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





César García Hernández

CONSIDERACIONES.

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

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

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

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

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

OBSERVACIONES:

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

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

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





LA NOMENCLATURA SOLICITADA PARA ENVIAR SU TRABAJO ES LA SIGUIENTE :

AAAA-MM-
DD_TNM_CELAYA_MATERIA_DOCUMENTO_[EQUIPO]_NOCTROL_APELLIDOS_NOMBRE_SEM.PDF

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

DONDE :

TNM_CELAYA	:	INSTITUCIÓN ACADÉMICA
AAAA	:	AÑO
MM	:	MES
DD	:	DÍA
MATERIA	:	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

César García Hernández

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





César García Hernández

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.

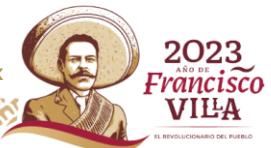
SE LE SUGIERE ENVIAR CON ANTICIPACIÓN SU ACTIVIDAD A FIN DE EVITAR CONFLICTOS POR NO ENTREGAR ÉSTA A TIEMPO.

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.



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



TECNOLÓGICO NACIONAL
DE MÉXICO EN
CELAYA

Asignatura: LENGUAJES & AUTOMÁTAS II
Docente: I.S.C. Ricardo González González

2023-10-02 LA-II-A

Actividad: Monografía: ANÁLISIS SINTÁCTICO
Aspectos esenciales para desarrollarlo e implementarlo

EQUIPO #1

INTEGRANTES:

- ▷ Arroyo Gómez José Alfredo (20030029)
- ▷ García Hernández Cesar (20030855)
- ▷ Gasca Palacio Jesús Fernando (20030606)
- ▷ González Mancera Christian Manuel (20030115)

TABLA DE CONTENIDO

INTRODUCCIÓN	1
GENERALIDADES	2
Objetivos.....	2
DESARROLLO.....	3
Tema 3.3 Precedencia de operadores.....	3
Tema 3.4.1 Analizador Sintáctico Descendente	10
Tema 3.4.2 Analizador Sintáctico Ascendente	16
Tema 3.5 Diseño y aplicación de una tabla de símbolos.	24
Tema 3.6 Manejo de errores sintácticos y su recuperación.....	33
Tema 3.6 Generadores de código para analizadores sintácticos	38
REFERENCIAS BIBLIOGRÁFICAS	40
Ejercicios Prácticos: YACC BISON.	43
Programa 1.....	47
Programa 2	50
Programa 3	54
CONCLUSIONES	63
TAREA: Análisis video: Monopolio de ASML – Importancia de los CPU	65
ENLACE VIDEO ACTIVIDAD 5.....	71

Título: ANÁLISIS SINTÁCTICO: aspectos esenciales para desarrollarlo e implementarlo

INTRODUCCIÓN

Como ya sabemos, los compiladores son programas que reciben el código fuente y nos regresan un archivo objeto. A lo largo de este proceso, nos encontramos con varias fases que cumplen una función específica y que reciben una entrada y tienen una salida.

Estas fases son:

- Análisis Léxico
- Análisis Sintáctico
- Análisis Semántico
- Código Intermedio
- Optimización

La fase en la que está centrado este documento es precisamente en la parte sintáctica; es decir, cual es la gramática y las reglas que estructuran a los lenguajes de programación. Siempre que vemos un error del tipo: "Error de sintaxis en la línea n, se esperaba X en vez de Y", estamos hablando de errores de sintaxis que se cometen u ocurren debido a que no estamos cumpliendo con las reglas establecidas dentro del lenguaje definido. Sin este tipo de reglas que controlen la estructura del lenguaje, nosotros no tendríamos la capacidad de formalizar un lenguaje, ya que no tendría un orden, un control o una forma. Es por ello por lo que, en este presente documento, nos encuendramos a la tarea de realizar una investigación en la cual presentamos nuestros hallazgos para reforzar y crear nuevas conexiones neuronales que representan nuestro conocimiento en el tema.

Los temas a tratar son los siguientes:

- 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

En el desarrollo de estos temas también incluimos ilustraciones, conceptos, ejemplos y ejercicios o prácticas en las que aplicamos los temas estudiados.

GENERALIDADES

Objetivos

- Comprender la importancia del análisis sintáctico y cuáles son sus principales características.
- Conocer el manejo de la precedencia de operadores a nivel de análisis sintáctico.
- Aplicar y estudiar estructuras de datos como los árboles, enfocados al análisis sintáctico.
- Estudiar el diseño de las tablas de símbolos.
- Conocer cómo es posible manejar los errores sintácticos.
- Repasar el concepto de generadores de analizadores sintácticos.
- Realizar ejercicios prácticos con YACC, BISON (Generadores de analizadores sintácticos).

DESARROLLO

TEMA 3.3 PRECEDENCIA DE OPERADORES

La precedencia de operadores es un tema recurrente en las matemáticas y para nosotros también, ya que lo hemos estudiado a lo largo de nuestra vida escolar. La precedencia de operadores nos permite definir el orden en que se deben leer las expresiones y operaciones para saber cómo operan, precisamente, los operandos junto con las constantes o literales, para así hallar el resultado correcto.

Algunos conceptos importantes son:

Operador: Símbolo o carácter que se asocia a una función o tipo de cálculo matemático que realiza en base a su(s) operando(s).

Operando: Puede ser una variable, literal, constante número, etc. El cuál actúa sobre un operador.

$$5 + 3 = 8 \rightarrow \text{Resultado}$$

Diagrama que muestra la descomposición de la operación $5 + 3 = 8$. Se ilustran los siguientes componentes:

- Operando:** Los números 5 y 3, que son los valores que se combinan.
- Operador:** El símbolo '+' que indica la adición.
- Resultado:** El número 8, que es el resultado final de la operación.

CONCLUSIONES

Tipos de operadores

Existe una gran variedad de tipos de operadores que cumplen con una función específica. Algunos de estos son:

- Operadores aritméticos → Realizan cálculos matemáticos de 1 o más números.
- Operadores de comparación → Comparan características específicas entre operandos.
- Operadores lógicos → Complementan a los operadores de comparación ampliando la expresividad de estos.
- Operadores de concatenación → Permiten unir cadenas.
- ETC (Depende del lenguaje)

A pesar de la diversidad, entre los operadores resaltan 3 características: Aridad, Asociatividad y Precedencia

1. Aridad: Se refiere a la cantidad de operandos que requiere el operador como mínimo. Por ejemplo: La suma tiene aridad 2 → $1 + 7$

2. Asociatividad: Se refiere a la dirección en que se debe leer y ejecutar la operación. De izquierda a derecha o de derecha a izquierda.

3. Precedencia: Sinónimo de prioridad. Indica que operandos se ejecutan primero antes que otros.

Prioridad De Los Operadores

Los distintos tipos de operadores tienen su propia prioridad o precedencia que ayuda a identificar el orden de una expresión completa. Veamos a continuación la prioridad de los operadores de C:

Símbolo	Tipo de operación	Asociatividad
[] () . → ++ -- (postfijo)	Expresión	De izquierda a derecha
sizeof & * + - ~ ! ++ -- (prefijo)	Unitario	De derecha a izquierda
typecasts	Unitario	De derecha a izquierda
* / %	Multiplicativo	De izquierda a derecha
+ -	Aditivo	De izquierda a derecha
<< >>	Desplazamiento bit a bit	De izquierda a derecha
< > <= >=	Relacional	De izquierda a derecha
== !=	Igualdad	De izquierda a derecha
&	AND bit a bit	De izquierda a derecha
^	OR exclusivo bit a bit	De izquierda a derecha
	OR inclusivo bit a bit	De izquierda a derecha
&&	AND lógico	De izquierda a derecha

Símbolo	Tipo de operación	Asociatividad
	OR lógico	De izquierda a derecha
? :	Expresión condicional	De derecha a izquierda
= *= /= %= += -= <= >= &= ^= =	Asignación simple y compuesta	De derecha a izquierda
,	Evaluación secuencial	De izquierda a derecha

Tabla de prioridad y asociatividad de los operadores de C

<https://learn.microsoft.com/es-es/cpp/c-language/precedence-and-order-of-evaluation>

Como podemos observar en la tabla anterior, se presentan reglas que determinan que operadores deben actuar primera sobre los operandos, empezando por paréntesis o corchetes, por ejemplo, pasando por operadores de prefijo unario, la multiplicación, la adición, desplazamiento de bits, etc.

Okey, esto es completamente entendible ya que es algo que nos enseñan a lo largo de nuestra vida escolar. Pero ¿Cómo nosotros podemos indicarle al computador que pueda entender la prioridad de un operador sobre otro? Que el computador pueda asociar el orden correcto de los operadores sin que nosotros lo digamos explícitamente, operador a operador. Bueno la solución es una de las más grandiosas estructuras de datos: Los árboles

Árboles

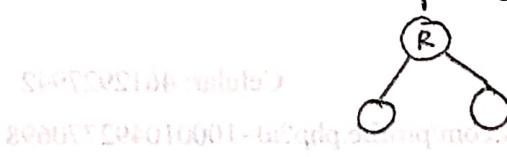
Más específicamente necesitaremos de los árboles sintácticos, los cuales, son un tipo de árbol especial que nos puede ayudar a representar la prioridad de los operadores gracias a que es una estructura de datos práctica, versátil y eficiente.

Conceptos

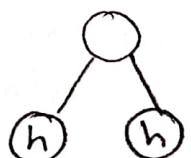
Nodos: Unidad esencial de un árbol. A, B, C, ..., +, 1, ...

Etiquetas: Identificación de un nodo. A, B, C, ..., +, 1, ...

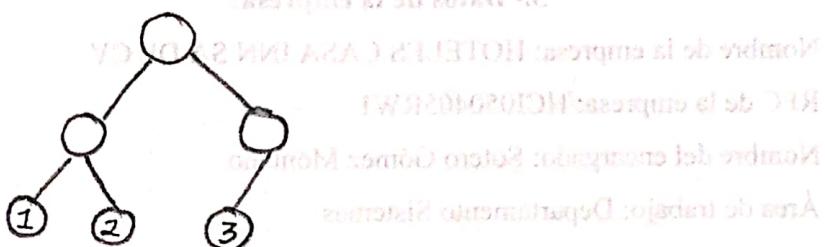
Nodo raíz: Es el nodo superior que es padre de todos los demás nodos inferiores. Es el primer nodo que ingresa al árbol.



Nodos hijos: Nodos que preceden de otro nodo que es superior a estos.



Nodos terminales: Nodos que no contienen hijos. En los árboles sintácticos los nodos terminales son valores numéricos.



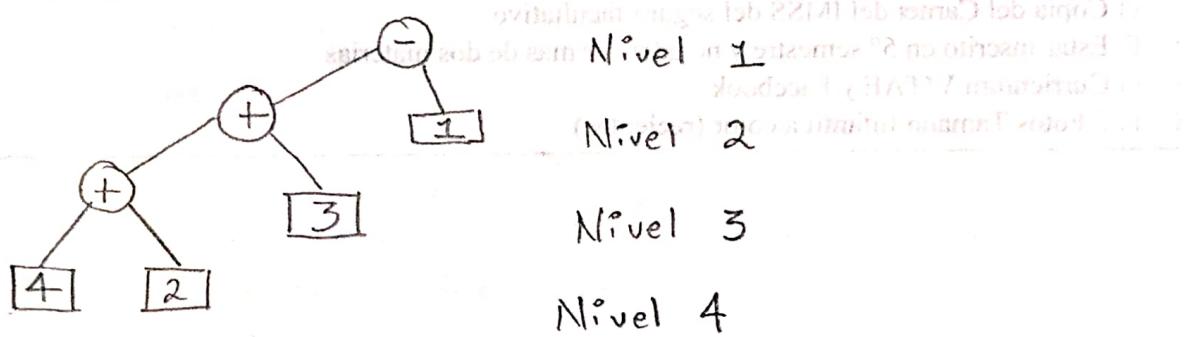
Árbol: Conjunto de nodos interconectados. Existe un nodo raíz y puede tener hijos.

Árbol binario: Cada nodo tiene exactamente 2 hijos o 0.

En los árboles sintácticos los nodos no terminales o padres indican el operador.

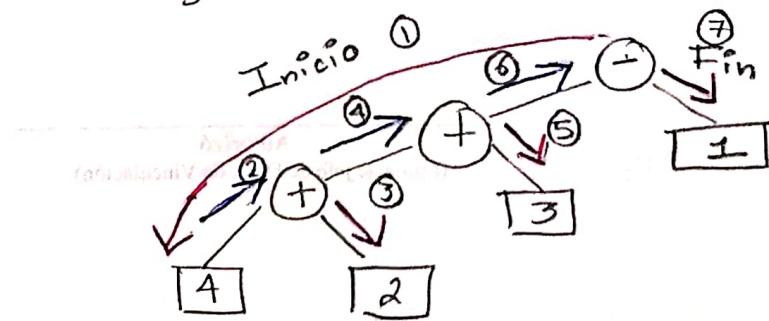
Ejemplos

Teniendo la expresión $4 + 2 + 3 - 1$ podemos representarla con un árbol agregando los operadores como nodos padres ordenados por su prioridad, los nodos con mayor prioridad van más abajo en cuestión de niveles. Los nodos terminales serán los números.



Si leemos la expresión de izquierda a derecha primero meteríamos un nodo terminal, un nodo "operador", un nodo terminal, un nodo "operador" y así según la expresión para completar la figura.

Si queremos obtener la expresión de un árbol debemos realizar una búsqueda en profundidad. Para ello el recorrido es el siguiente:



1. = 4
2. = 4 +
3. = 4 + 2
4. = 4 + 2 +
5. = 4 + 2 + 3
6. = 4 + 2 + 3 -
7. = 4 + 2 + 3 - 1

Cuando nos encontramos con expresiones más complejas es importante seguir la siguiente estrategia:

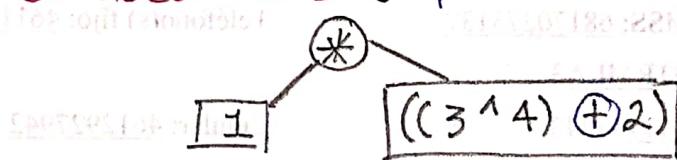
- Balancear la expresión
 - Tomar el nodo operador del centro como raíz
- Repetir el proceso (En lugar de raíz, el nodo operador sera padre).

Expresión: $1 * 3^4 + 2$

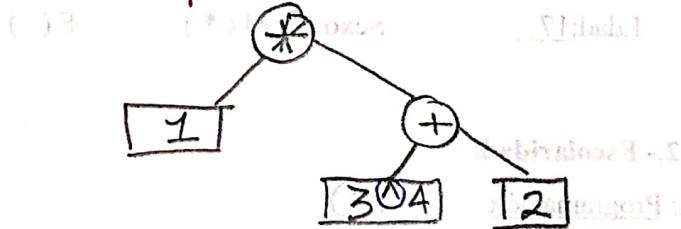
Balanceamos con parentesis (indicamos las prioridades)

$1 * ((3^4) + 2)$

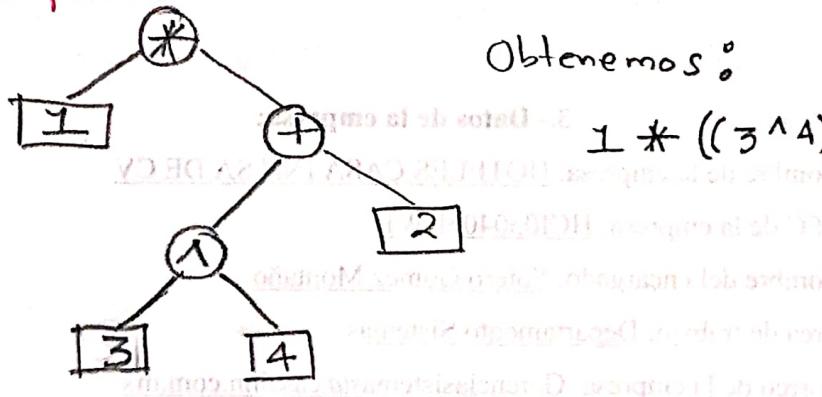
Colocamos el nodo raíz (operador de en medio)



Colocamos el nodo padre



Colocamos el nodo aparente



Obtenemos:

$$1 * ((3^4) + 2)$$

Podemos observar que es muy importante indicar el orden en que almacenamos los nodos para representar la operación que queremos realizar en realidad.

ANALIZADOR SINTÁCTICO DESCENDENTE

En investigaciones previas, hemos mencionado el análisis sintáctico, que en el contexto de la materia de Lenguajes y Autómatas II encontramos uso por ejemplo en compiladores pues es el encargado de llevar a cabo la verificación y revisión de una cadena de entrada en base a la gramática comprobando que pertenezca a un lenguaje, para después "alimentar" al árbol sintáctico.

Una vez se hace comprensión sobre este concepto, podemos enfocarnos por completo en el analizador sintáctico descendente, que cumple con la función de suministrar al árbol sintáctico mencionado previamente, pero mediante algunas pautas:

- Regirse en las reglas básicas de la estructura y funcionamiento del lenguaje.
- Decidir cómo aplicar las reglas gramaticales.
- Llevar a cabo Left Most Derivation.
- Generar el árbol de análisis sintáctico.

Básicamente existen dos tipos de análisis sintáctico descendentes: analizador sintáctico por descenso recursivo y analizador sintáctico predictivo.

ANALIZADOR SINTÁCTICO POR DESCENSO RECURSIVO

Estos son un conjunto de procedimientos recursivos, es decir, que si una regla no se cumple como se esperaba

se procede a retroceder y se lleva a cabo la prueba con la siguiente regla.

Cada elemento no terminal en la gramática se asocia con un procedimiento específico. Durante la construcción del árbol de análisis, cuando se llega a un terminal, se verifica si el procedimiento asociado es el correcto. Si es el correcto, se avanza al siguiente token. En caso contrario, se genera un mensaje de error y se aplica una estrategia de recuperación. Esta estrategia puede implicar deshacer la operación actual, retroceder y llamar a otro procedimiento. De ahí la recursividad.

Además se puede ver como un esfuerzo por crear un árbol de análisis sintáctico para la entrada desde el punto de partida y estableciendo los nodos del árbol en un orden específico.

Ejemplo

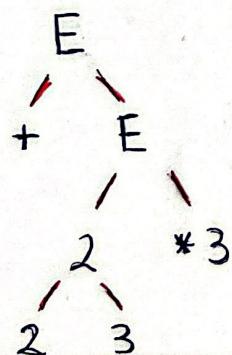
Teniendo la siguiente gramática:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid \text{NUM}$$

Donde entendemos que una expresión puede ser: la suma, diferencia, producto o cociente de dos expresiones o un número.

Se suministra la siguiente cadena de entrada: $5 + 2 * 3$.

El árbol se construye de izquierda a derecha, comenzando con el símbolo inicial E .



1. Se utiliza la primera regla para derivar $E+E$.
2. Se utiliza la segunda regla para derivar $2 \times E$.
3. Se utiliza la tercera regla para derivar 2×3 .
4. Se utiliza la cuarta regla para derivar $5+2 \times 3$.

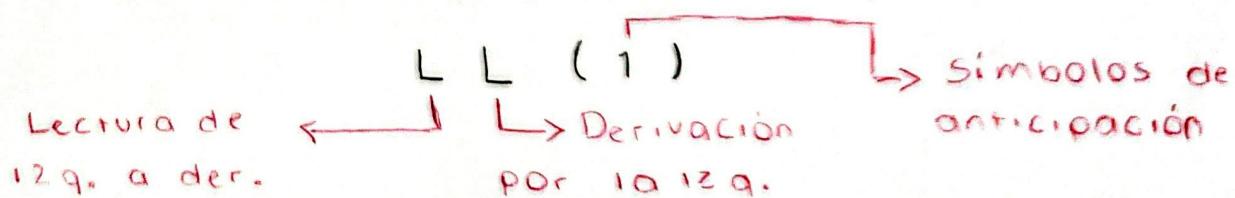
ANALIZADOR SINTÁCTICO PREDICTIVO

Este es un tipo de programa que ayuda a entender cómo funcionan las oraciones en un lenguaje, y lo hace sin tener que retroceder. Para que funcione, es importante que, a partir de la primera palabra que queremos entender, podamos saber qué regla va a ser usada.

El método LL(1) es efectivo porque se basa en un tipo de máquina que contiene tanto palabras clave como partes de una oración. También utiliza una tabla especial que nos dice qué hacer con cada palabra que queremos entender y con qué parte empezar.

Entonces, se determina qué regla de producción se usará dependiendo del token que se tenga en el instante de lectura de la cabeza.

Las siglas LL(1) tienen un significado que se explica a continuación:



Para utilizar el análisis descendente predictivo LL(1), es esencial emparejar cada regla de producción con un conjunto de predicción. El conjunto de predicción de una regla se compone de una lista que incluye todos los términos posibles que deben aparecer al principio

de una lectura para que esa regla pueda ser aplicada. Estos conjuntos se construyen haciendo apoyo de dos conjuntos referente a la parte izquierda y derecha de las reglas.

Para que se efectúe exitosamente LL(1) la gramática tiene que cumplir 3 requerimientos:

- No ambigüedad: esto implica que la gramática usada para describir un lenguaje no debe tener dobles interpretaciones. En otras palabras, cada frase o estructura en el lenguaje debe tener una sola forma de entenderse. La ambigüedad puede causar confusión y problemas al analizar el lenguaje de programación.
- Organización de izquierda a derecha: este requerimiento se refiere a cómo se estructuran las reglas gramaticales. Las entradas se leerán de izquierda a derecha en el proceso de análisis, lo que ayuda a simplificar el proceso y ayuda a evitar posibles conflictos.
- No recursión a izquierda: la recursión a izquierda se refiere a la situación en la que una regla gramatical se llama a sí misma desde el lado izquierdo de la producción. Se evita debido a que puede generar ambigüedad y complicar la construcción que brinda un analizador sintáctico predictivo eficiente. En su lugar, se prefiere la recursión a la derecha, donde una regla se llama a sí misma desde el lado derecho de la producción.

El método LL(1) se resume en tres características principales mencionadas por la Universidad Europea de Madrid:

1. No necesita retroceso.
2. Se sabe qué producción hay que aplicar a cada token.
3. No puede haber ambigüedad en la gramática.

Ejemplo

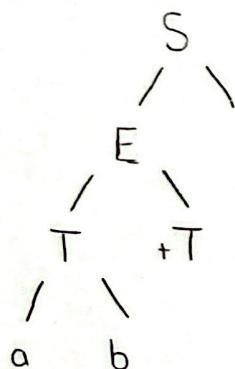
Considerando la siguiente gramática:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow a \mid b \end{aligned}$$

Esta gramática define un lenguaje que consiste en cadenas de caracteres que representan expresiones aritméticas sencillas. Como las siguientes cadenas:

$$\begin{array}{c} a+b \\ a+b+a \end{array}$$

Si usamos la cadena de entrada $a+b$ como base, el árbol de derivación se vería de la siguiente manera:



Las reglas de producción usadas para la construcción del árbol de derivación son las siguientes:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow T + T \\ T &\rightarrow a \\ T &\rightarrow b \end{aligned}$$

Donde el proceso fue el siguiente:

Se comienza con la regla $S \rightarrow E$ en la cima de la pila, se lee el primer símbolo de la cadena de entrada. Después

se aplica la regla de producción que tiene el símbolo leído como primer símbolo del lado derecho. Posteriormente se aplica la regla de producción y se coloca el símbolo leído en la pila. Se repiten los pasos mencionados hasta que se hayan leído todos los símbolos de la cadena de entrada.

AUTÓMATA A PILA

Sin importar el tipo de analizador sintáctico empleado, todos utilizan un componente denominado autómata a pila (también se conoce como autómata con pila o basado en pila) en su proceso de análisis.

Este componente se utiliza para examinar una entrada y determinar si cumple con las reglas gramaticales. Tanto los analizadores sintáticos descendentes como los ascendentes se apoyan en este mecanismo o modelo para evaluar la entrada en cuestión.

Los autómatas a pila mantienen una estructura equivalente a la de los autómatas finitos, pero incorporan a una adicional que funciona como una especie de memoria auxiliar. Esta pila se emplea para almacenar información relevante que se requerirá durante el proceso de análisis.

La manera formal de definir un autómata a pila es la siguiente.

$$AP = (\Sigma, \Gamma, Q, A_0, q_0, f, F), \text{ donde:}$$

- Σ : alfabeto de símbolos de entrada.
- Γ : alfabeto de símbolos de la pila.
- Q : es el conjunto de estados.
- A_0 : es el símbolo inicial de la pila.
- q_0 : es el estado inicial y forma parte de Q .

- f : es la función de transición entre estados.
- F : es el conjunto de estados finales.

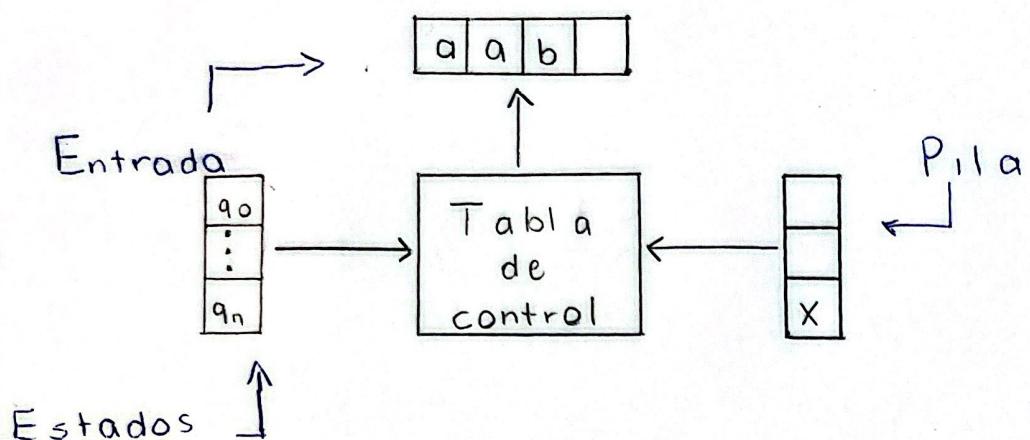


Ilustración. Autómata a pila.

Un autómata a pila es la forma de representar un lenguaje basado en una gramática independiente del contexto, del mismo modo que las expresiones regulares utilizan un autómata finito determinista.

3.4.2 Analizador Sintáctico Ascendente

Al hablar sobre el tema de análisis sintáctico encontramos el subtema del analizador sintáctico ascendente, también puede ser conocido como análisis sintáctico por desplazamiento y reducción, este intenta construir un árbol de análisis sintáctico para una cadena de entrada que comienza por las hojas o desde el fondo (la parte inferior) y va avanzando hasta la raíz o la cima del árbol (la parte superior).

Podemos nombrar a este proceso como "reducir" una cadena w hasta el símbolo inicial de una gramática.

De manera general el proceso se lleva a cabo donde cada paso del proceso de reducción se sustituye una cadena que concuerde con el lado derecho de la producción por el símbolo del lado izquierdo de la producción, así en cada paso se debe elegir de manera correcta la cadena, y trazar una derivación por la derecha en sentido inverso.

Características:

- Se aplican reducciones (pasos de la derivación en reversa) y desplazamientos (lecturas de tokens de entrada).
- Las hojas del árbol sintáctico son los símbolos terminales de entrada (tokens) y el nodo raíz es el símbolo inicial.
- Es uno de los métodos más poderosos y generales.
- Explora la secuencia de tokens de izquierda a derecha.

Podemos ver que el análisis sintáctico ascendente es una técnica que intenta comprobar si una cadena x pertenece al lenguaje definido por la gramática $L(G)$ aplicando lo siguiente

- Partir de los elementos terminales de la frase x
- Escoger reglas gramaticales estratégicamente
- Aplicar a la inversa derivaciones por la derecha
- Procesar la cadena de izquierda a derecha.
- Intentar alcanzar el axioma para obtener el árbol de análisis sintáctico o error.

Ejemplo $\rightarrow R_1 : E ::= E + E$

$R_2 : E ::= E - E$

$R_3 : E ::= E * E$

$R_4 : E ::= E / E$

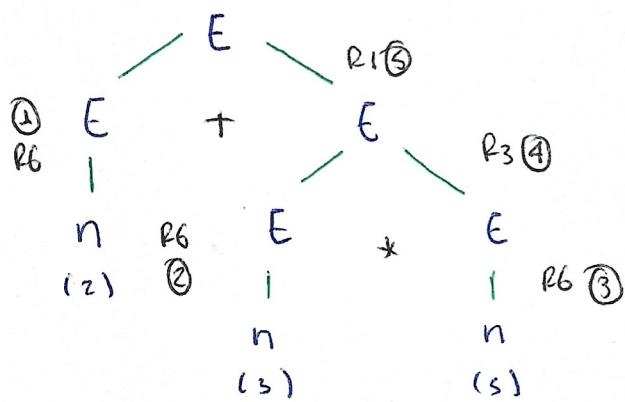
$R_5 : E ::= (E)$

$R_6 : E ::= n$

Cadena de derivación \rightarrow El análisis ascendente genera una cadena de derivación por la derecha leída en sentido inverso.

$n + n * \leftarrow$
 $E + n * n \leftarrow$
 $E + E * n \leftarrow$
 $E + E * E \leftarrow$
 $E + E \leftarrow$
 E

Árbol de análisis sintáctico \rightarrow Se parte la cadena de entrada leída de izquierda a derecha y se aplican reglas a la inversa para intentar alcanzar el axioma.



Otras formas de representación pueden ser las siguientes:

→ Considere la gramática

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \mid b \\ B &\rightarrow d \end{aligned}$$

La frase abbcde se puede reducir a s por los siguientes pasos:

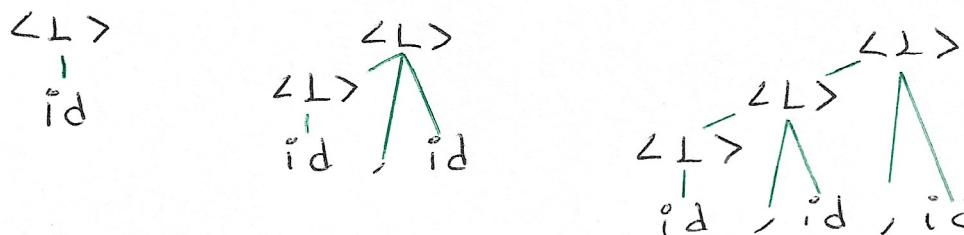
abbcde
aAbcde
aAde
aABe
S

Se examina abbcde buscando una subcadena que concuerde con el lado derecho de alguna producción. Las sub cadenas b y d sirven. Elige se la b situada más a la izquierda y sustituyase por A, el lado izquierdo de la producción $A \rightarrow b$; así que se obtiene la cadena aAbcde. A continuación las sub cadenas Abc, b y d concuerdan con el lado derecho de alguna producción. Aunque b es la sub cadena situada más a la izquierda que concuerda con el lado derecho de una producción, se elige sustituir la sub cadena Abc por A, que es el lado derecho de la producción $A \rightarrow Abc$. Se obtiene ahora aAde.

Sustituyendo después d por B, que es el lado izquierdo

de la producción $B \rightarrow d$, se obtiene a ABe. Ahora se puede sustituir toda esta cadena por s. Por lo tanto, mediante una secuencia de cuatro reducciones se puede reducir abcede a s. De hecho, estas reducciones trazan la siguiente derivación por la derecha en orden inverso: $S \xrightarrow{m_2} aABe \xrightarrow{m_2} aAde \xrightarrow{m_2} aAbcd \xrightarrow{m_2} abbcd$.

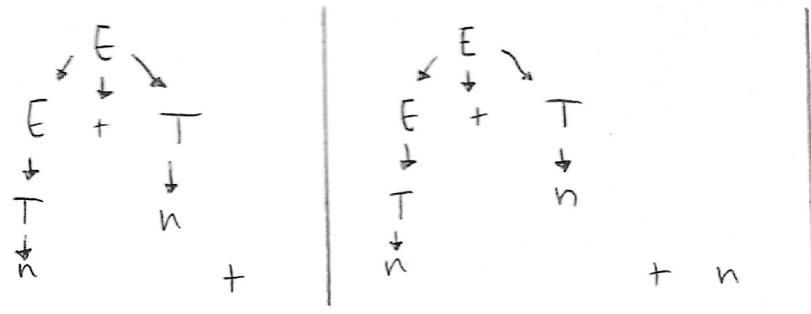
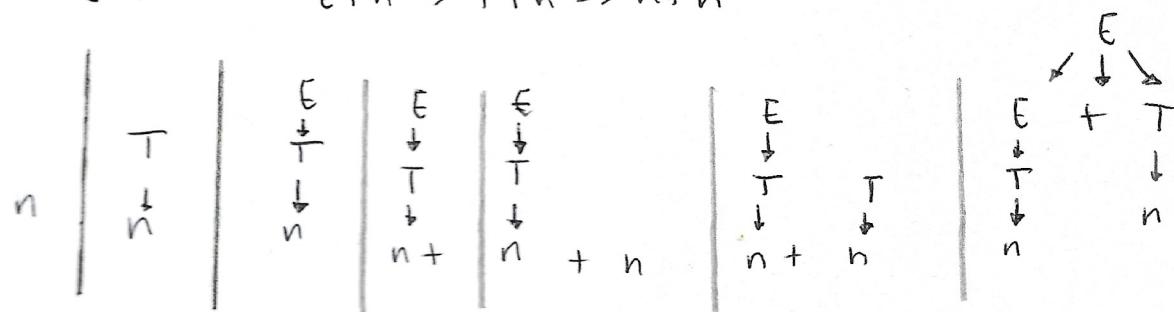
→ Gramática $\langle L \rangle \rightarrow id$
 $\langle L \rangle \rightarrow \langle L \rangle, id$



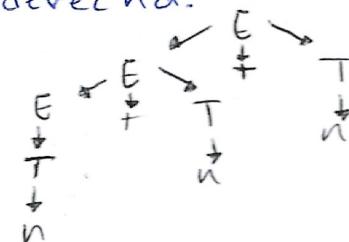
→ Gramática $G' = (\{E, T\}, \{n, +\}, P, E)$:
 $E \rightarrow T \mid E + T$
 $T \rightarrow n$

y la cadena de entrada $n+n+n$ el proceso de análisis sintáctico ascendente podría ser:

- $E \Rightarrow E + T \Rightarrow E + n \Rightarrow T + n \Rightarrow n + n$



observar que la derivación obtenida es más a la derecha.



Es importante conocer que al usar este análisis el problema fundamental es decidir cuando lo que parece ser la parte derecha de una regla puede ser sustituida por la parte izquierda.

Y que no es un problema trivial ya que pueden existir ocasiones en las que es posible sustituir dos producciones diferentes.

Análisis Sintáctico por desplazamiento y reducción.

El análisis ascendente lineal más utilizado es el algoritmo de desplazamiento-reducción (shift-reduce).

Este algoritmo se basa en una tabla de análisis y una pila de estados.

Existen diferentes métodos para generar la tabla de análisis ($LR(0)$, SLR , $LALR$, $LR(1)$).

El método $LR(1)$ genera la tabla para cualquier gramática $LR(1)$ pero genera tablas muy grandes.

El método SLR genera tablas muy compactas pero no puede aplicarse a todas las gramáticas $LR(1)$.

El método $LALR$ genera tablas compactas y puede aplicarse a la mayoría de gramáticas $LR(1)$.

$LALR$ es una modificación del método $LR(1)$, llamado Método $LALR(1)$ mantiene el poder del $LR(k)$ y preserva la eficacia de $SLR(1)$.

El análisis (ALR(1)) se basa en la observación de que en muchos casos el tamaño grande del AFD de ítems LR(1) se debe a la existencia de muchos estados diferentes que tienen igual la primera componente, los ítems LR(0), y difieren solo de la segunda componente, los símbolos de preanálisis.

Utiliz dos acciones básicas

- Desplazar: consiste en consumir un token de la cadena de entrada.
- Reducir: consiste en sustituir en la pila de símbolos de una parte derecha de una regla por su parte izquierda.

Otras acciones

- Aceptar: terminar el análisis aceptando la cadena,
- Error: producir un error (cuando no se encuentra ninguna acción).

Esquema análisis sintáctico ascendente.

① SLR ; ③ LALR ; ④ LR(1)

Determinan la regla a aplicar
consultando el primer terminal a
la entrada

② LR(0)

Determinan la regla que
hay que aplicar sin consultar
terminales a la entrada

⑤ LR(K)

Determinan la regla a
aplicar consultando los
K primeros terminales
a la entrada

Análisis por reducción - desplazamiento

Se llevan a cabo 2 operaciones, a cada paso el analizador decide realizar una operación.

Desplazar → El analizador procesa terminales desplazando la cabeza lectora antes de aplicar una regla de producción a la inversa.

Reducir → Cuando se reconoce en la forma de frase en curso una parte igual a la parte derecha de una regla, ésta se sustituye por el antecedente.

$$R_1 - E ::= E + E \quad R_3 - E ::= E * E \quad R_5 - E ::= (E)$$

$$R_2 - E ::= E - E \quad R_4 - E ::= E / E \quad R_6 - E ::= n$$

—mango d - desplazar r - reducir

$$\begin{matrix} n & + & n & * & n \\ \Delta & & & & \\ d & & & & \end{matrix}$$

$$\begin{matrix} \text{init} & n & * & n \\ \Delta & & & \\ r & & & \end{matrix}$$

$$\begin{matrix} E & + & n & * & n \\ \Delta & & & & \\ d & & & & \end{matrix}$$

$$\begin{matrix} E & + & n & * & n \\ \Delta & & & & \\ d & & & & \end{matrix}$$

$$\begin{matrix} E & + & \text{init} & * & n \\ \Delta & & & & \\ r & & & & \end{matrix}$$

$$\begin{matrix} E & + & E & * & n \\ \Delta & & & & \\ d & & & & \end{matrix}$$

$$\begin{matrix} E & + & E & * & n \\ \Delta & & & & \\ d & & & & \end{matrix}$$

$$\begin{matrix} E & + & E & * & \underline{\underline{n}} \\ \Delta & & & & \\ r & & & & \end{matrix}$$

$$\begin{matrix} E & + & E & * & E \\ \Delta & & & & \\ r & & & & \end{matrix}$$

$$\begin{matrix} E & + & E \\ \Delta & & \\ r & & \end{matrix}$$

E

También podemos encontrar 2 tipos de conflictos

Reducción-desplazamiento

→ El analizador puede tanto aplicar un paso de reducción como uno de desplazamiento.

Reducción-reducción

→ El analizador puede aplicar 2 reglas distintas para realizar diferentes pasos de reducción.

$$n + n \star n$$

d

$$\boxed{n} + n \star n$$

r

$$E + n \star n$$

d

$$E + n \star n$$

d

$$E + \boxed{n} \star n$$

r

$$\boxed{E + E \star n}$$

d

$$E + E \star n$$

d

$$\boxed{E + \boxed{E \star E}}$$

r

$$\boxed{\underline{E} + \underline{E}}$$

r

E

DISEÑO Y ADMINISTRACIÓN DE UNA TABLA DE SÍMBOLOS

Definición de una tabla de símbolos

En los lenguajes de programación, el diseño y la administración de una tabla de símbolos es esencial para lograr un funcionamiento eficiente y efectivo.

Podemos definir una tabla de símbolos como una estructura de datos que se utiliza en la etapa de compilación de un lenguaje, es específico en un compilador. Esta estructura de datos almacena y gestiona la información sobre los identificadores utilizados en el código fuente, a fin de llevar un registro de la información sobre el ámbito y el enlace de los nombres.

Estos nombres o identificadores pueden ser desde variables, constantes, funciones u otros elementos que requieren ser identificados y asociados con información adicional, esta información se detallara mas adelante.

Así pues podemos definir los fines de una tabla de símbolos en:

- Almacenamiento de todos los identificadores de forma estructurada.
- Verificar cada variable.
- Comprobar tipo de implemento, comprobando las expresiones en el código fuente (semántica).

- Determinar el alcance de cada identificador.

De esta manera podemos definir una tabla de símbolos como; una tabla que mantiene una entrada para cada uno de los nombres o identificadores.

Objetivos de una Tabla de símbolos

Como se menciona anteriormente las tablas de símbolos son estructuras de datos que almacenan toda la información sobre los identificadores del lenguaje fuente.

Como tal debe cumplir con estos objetivos:

- Almacenar información sobre los identificadores utilizados en el código fuente, como variables, funciones y tipos de datos.
- Facilitar la búsqueda rápida y eficiente de información relacionada con los símbolos durante la compilación.
- Garantizar la unicidad de los nombres de símbolos, evitando conflictos y ambigüedades en el lenguaje.
- Ayudar en la gestión de ámbito o alcances de los símbolos, asegurando la adecuada resolución de variables locales y globales.
- Registrar información sobre el tipo de datos de los símbolos.
- Proporcionar información para la generación de código.

Todos estos objetivos dependen de las características del lenguaje, se obtienen directamente del análisis del programa fuente, aunque en ocasiones se pueden obtener a través del contexto en el que aparece el símbolo.

Diseño de una tabla de símbolos

El diseño de una tabla de símbolos es fundamental en la construcción de un compilador o intérprete. Debe contemplar diversos aspectos para garantizar su eficiencia y utilidad durante el proceso de compilación. A continuación mencionaremos algunos aspectos clave a considerar durante su diseño.

► Estructura de almacenamiento

La estructura de almacenamiento en una tabla de símbolos es clave a la hora de representar estas. Se puede implementar diferentes estructuras de datos; cada uno con características especiales, teniendo ventajas y desventajas en términos de velocidad de acceso y eficiencia de búsqueda, a continuación mencionaremos los más comunes:

*Tabla Hash: Es una de las estructuras más populares para implementar tablas de símbolos. En este enfoque se utiliza una función de hash, la cual es una función criptográfica, es decir un algoritmo matemático que transforma cualquier bloque arbitrario de datos en una serie de caracteres con una longitud fija.

De esta manera se le asigna un valor numérico a cada símbolo en función de su nombre. Estos valores numéricos se utilizan como índices o identificadores en una matriz, donde cada entrada contiene información sobre el símbolo correspondiente. Las tablas hash son eficientes para búsquedas rápidas, pero requieren manejo de colisiones.

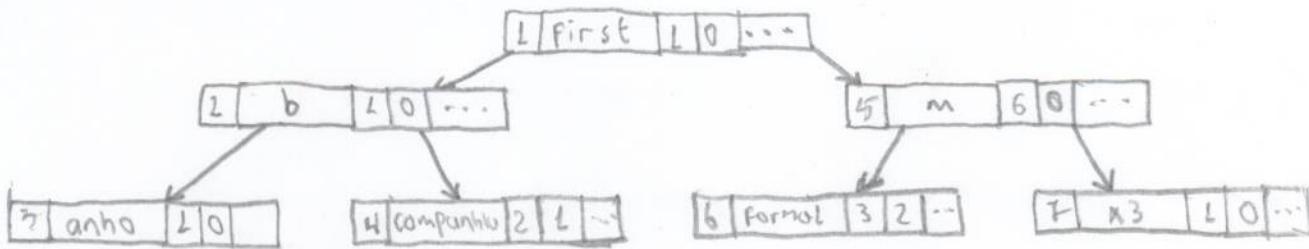
Nombre de la variable	
Posición	Largo
0	9
9	2



*Árbol de Búsqueda Binaria: Esta es otra estructura común. Sigue la estructura de árboles binarios, es decir de dos hijos. En este caso, los símbolos se almacenan en un árbol donde cada nodo tiene dos hijos, uno para símbolos mayores y otro para símbolos menores. Esto permite búsquedas eficientes, especialmente si el árbol se balancea correctamente (mismo número o nivel del árbol respecto a los nodos). Sin embargo, la inserción y eliminación puede ser menos eficiente que una tabla hash.

Posición	ID	Tipo	Otros atributos
1	first	L	...
2	b	L	...
3	anho	L	...
4	compania	2	...
5	m	6	...
6	format	3	...
7	x3	L	...

Se ordena alfabéticamente, aunque varía dependiendo el lenguaje



*Árbol AVL o Árbol-Rojo-Negro: Estas estructuras son árboles binarios平衡ados que garantizan un rendimiento óptimo en operaciones de inserción y búsqueda.

*Tabla de Símbolos Anidadas: En lenguajes que admiten ámbitos anidados, como C o Python, es común utilizar una estructura de símbolos anidada. Cada ámbito tiene su propia tabla de símbolos y estas tablas se organizan en una estructura jerárquica que coincide con la jerarquía de ámbitos en el código fuente.

Como se puede ver cada estructura tiene ventajas y desventajas dependiendo del enfoque y selección de las necesidades específicas del compilador o intérprete.

► Contenido de la tabla

La tabla de símbolos contiene información referente a los tipos de datos creados, en general cualquier identificador creados por el usuario. La información que se decide almacenar en la tabla de símbolos dependerá de las características concretas del compilador o traductor que esté desarrollando. Entre esta información se encuentra:

* **Nombre del símbolo:** Es el nombre único que identifica al símbolo, como el nombre de una variable o función. Este nombre puede almacenarse limitando o no su longitud, si llega a limitarse un tamaño máximo fijo, se puede aumentar la velocidad de creación y búsqueda; pero esto puede desperdiciar o necesitar espacio según el caso.

(las búsquedas) en la tabla de símbolos suelen hacerse a través del nombre; por ello es más común usar tablas hash.

* **Tipo de símbolo:** Es el tipo de dato asociado al símbolo (entero, cadena, función, etc.). Resulta fundamental conocer el tipo de datos al que pertenece cada símbolo.

* **Dirección de memoria:** Es la dirección de memoria en la que se almacena el valor del símbolo en tiempo de ejecución. Esta dirección es importante, ya que las instrucciones que refieren a una variable o función deben saber donde encontrar el valor de esa variable para poder generar código máquina.

* **Valor:** Es el valor actual del símbolo si es una variable o constante. En los intérpretes dado que los tiempos de compilación y ejecución se solapan resulta fácil gestionar los símbolos si almacenamos sus valores en la tabla de símbolos. En un compilador, la tabla de símbolos no almacenan su valor.

* **Número de dimensiones:** Si la variable a almacenar es un arraylo, también

pueden almacenarse sus dimensiones. Aunque esta información se puede extraer de la estructura de tipos.

- * **Ambito:** Es el ámbito en el que el símbolo es válido como global o local a una función.
- * **Desplazamiento o dirección:** Nos ayuda con la ubicación en memoria del símbolo, es esencial en la compilación.
- * **Otra información adicional:** Cualquier otra información relevante, como información de depuración, etiquetas, etc.

Resolución de conflictos

En ocasiones, puede ocurrir que dos símbolos diferentes tengan el mismo nombre. Para evitar conflictos y garantizar la correcta asociación de los símbolos con su información correspondiente, se deben implementar mecanismos de resolución de conflictos.

- * **Ambito de variables:** En muchos lenguajes de programación, las tablas de símbolos se organizan en ámbitos. Si dos símbolos tienen el mismo nombre pero pertenecen a diferentes ámbitos, no genera conflicto. El compilador o intérprete simplemente busca en el ámbito más cercano y resuelve el símbolo.
- * **Ambito Local vs Global:** Cuando un símbolo existe tanto en el ámbito local como en el global, generalmente se da prioridad al ámbito local. El compilador busca primero en el ámbito local, si no lo encuentra, busca en el ámbito global.
- * **Sobrecarga de Funciones:** En algunos lenguajes, se permite la sobrecarga de funciones, lo que significa que varias funciones pueden tener el mismo nombre pero diferentes parámetros. La resolución de conflictos se basa en los parámetros proporcionados en la llamada a la función para determinar cuál de las funciones con el mismo nombre se debe ejecutar.

- ***Alias o Renombramiento:** En ciertos casos, se pueden usar alias o renombramientos para evitar conflictos. Esto implica asignar un nuevo nombre a uno de los símbolos en conflicto.
- ***Errores de ambigüedad:** Si un compilador o intérprete no puede resolver un conflicto de manera inequívoca, se genera un error de ambigüedad y se solicita al programador que aclare la referencia al símbolo.

En resumen, la resolución de conflictos en tablas de símbolos implica determinar de manera efectiva cuál de los símbolos con el mismo nombre se debe utilizar en un contexto particular. Un manejo adecuado de conflictos en tablas de símbolos es esencial para garantizar la correcta ejecución e integridad de los datos.

Administración de una tabla de símbolos

Una vez diseñada, la tabla de símbolos debe ser administrada de manera eficiente durante el proceso de compilación. Algunos aspectos clave de su administración son los siguientes:

► Inserción de símbolos y búsqueda

La inserción de símbolos en tablas de símbolos es un proceso esencial en la programación y la compilación de lenguajes de programación. Las tablas de símbolos pueden considerarse como la base de datos subyacente.

Cada vez que se declara una variable, se define una función o se crea un nuevo identificador en el código fuente, se inserta un símbolo correspondiente en la tabla.

Uno de los propósitos fundamentales de las tablas de símbolos es garantizar la integridad del código fuente. Cuando se realiza la inserción de símbolos se llevan a cabo comprobaciones importantes. Por ejemplo,

el sistema verifica si ya existe un símbolo con el mismo nombre en el mismo ámbito. Esto previene la redifinición inadvertida de variables o funciones.

Así pues las dos operaciones se realizan en puntos concretos, estas dos operaciones son la inserción y la búsqueda. Si la tabla de símbolos está ordenada, entonces la operación de inserción llama a un procedimiento de búsqueda para encontrar el lugar donde colocar los atributos del identificador a insertar. En estos casos la inserción lleva tiempo,

Si no se encuentra ordenada, simplifica más el proceso de inserción, pero la operación de búsqueda se complica ya que debe examinar toda la tabla. La operación de búsqueda se lleva a cabo en todas las referencias de los identificadores, excepto en su declaración. La información que se busca se usa en la verificación semántica y en la generación de código.

En la búsqueda se detectan los identificadores que no han sido declarados previamente emitiendo mensajes de error. Mientras en la inserción se detectan los identificadores que ya han sido previamente declarados.

► Funcionamiento y acceso a la tabla de símbolos

Para que las tablas de símbolos funcionen se necesita la asociación del nombre del símbolo con su información relevante: tipo, valor, ámbito, dirección de memoria.

Las tablas suelen ser implementadas como estructuras de datos por lo general como se vio el método más eficiente es el método hash, pero todo depende de las necesidades.

Ejemplos básicos

• Tabla de Símbolos en un compilador

- Nombre : "x" • Dirección : 1000 (ubicación en memoria)
- Tipo : Entero.
- Valor : 0 (inicializado)
- Ámbito : Global

* Tabla de Símbolos en un Lenguaje de Programación Python:

- Nombre del símbolo: "nombre"
- Tipo: Cadena de caracteres.
- Valor: "Juan"
- Ámbito: Local (dentro de una función).
- Información adicional: Variable utilizada para almacenar un nombre.

* Tabla de Símbolos para variables en C

```
struct SymbolTableEntry {
```

```
    char name[20], // Nombre de la variable  
    int type; // Tipo de dato (entero, flotante, etc)  
    int address; // Dirección de memoria asignada
```

```
};
```

```
// Ejemplo de una tabla de símbolos para variables
```

```
SymbolTable [50]; // Capacidad para 50 variables
```

```
int main()
```

```
// Insertar los símbolos en la tabla
```

```
strcpy (symbolTable [0].name, "x");  
symbolTable [0].type = INT;  
symbolTable [0].address = 1000;
```

```
// Acceso a símbolos
```

```
printf ("Nombre: %s, Tipo: %d, Dirección: %d\n",
```

```
    symbolTable [0].name,  
    symbolTable [0].type,  
    symbolTable [0].address);
```

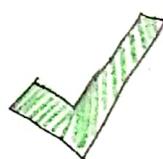
```
return 0;
```

4

TEMA 3.6 MANEJO DE ERRORES SINTÁCTICOS Y SU RECUPERACIÓN

Un aspecto común e interesante en el mundo de la programación son los errores que cometemos los programadores: faltó un punto y coma, un parentesis, se esperaba otro carácter, etc. Debido a que los humanos no somos perfectos y cometemos errores, los compiladores deben lidiar con estos desafíos. Si un compilador solo tuviera que lidiar con programas perfectos, todo sería más sencillo. Lastima que no es así el mundo real.

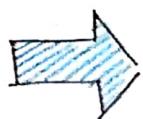
``
Print
('Perfect')
''



Correcto
Perfecto
Sí errores

Programa Perfecto
Imaginario

{} {}
imprimeme
un texto
por favor



Error en la línea
3 ("No hay nada
en la linea 3").

Mis programas
reales

A pesar de la frecuencia de los errores, no muchos lenguajes de programación han sido diseñados considerando cuidadosamente cómo manejarlos. La mayoría de las veces, cuando escribimos un programa incorrecto, el compilador debe ayudarnos a identificar y localizar estos errores.

Errores Sintácticos

Son aquellos que violan las reglas gramaticales del lenguaje, son los más fáciles de detectar y corregir.

¿Cómo manejar errores de sintaxis?

Los manejadores de errores en los analizadores sintácticos tienen objetivos claros:

- Informar sobre errores con precisión
- Recuperarse de los errores rápidamente
- No retrasar el procesamiento de programas válidos

En general los compiladores deberían de mostrar la línea de código fuente donde ocurre el error, esto puede ayudar a los programadores a entender dónde se equivocaron.

Existen compiladores que intentan reparar los errores por sí mismos. Sin embargo, esta recuperación completa de errores rara vez es perfecta y puede ser costosa en términos de recursos.

Estrategias de Recuperación de Errores

No existe una estrategia universalmente aceptada, pero, algunas de ellas son ampliamente aplicables y pueden ser bastante efectivas.

1. Recuperación en Modo Pánico

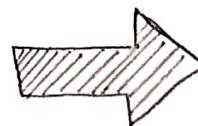
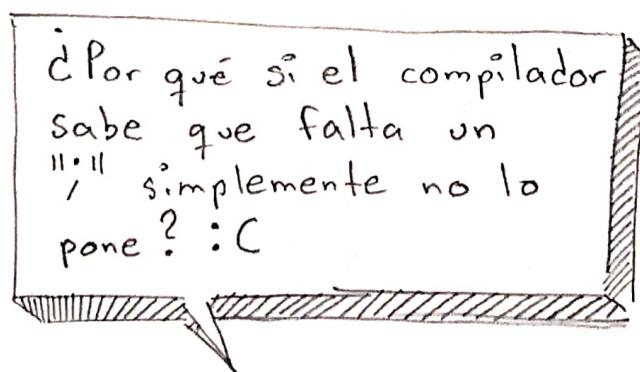
Esta estrategia es la más simple de implementar y es utilizada por la mayoría de los métodos de análisis sintáctico. Cuando se detecta un error, el analizador sintáctico descarta símbolos de entrada uno a uno hasta encontrar un símbolo que pertenezca a un conjunto predefinido de componentes léxicos de sincronización. Estos componentes léxicos de sincronización suelen ser delimitadores claros en el código, como punto y coma o palabras reservadas como "end". Aunque esta estrategia a menudo ignora una parte considerable de la entrada sin verificar si hay más errores, es simple y garantiza que el analizador no se quedará atrapado en un bucle infinito. Es especialmente adecuada cuando los errores múltiples en una misma proposición son raros.

2. Recuperación a nivel de frase

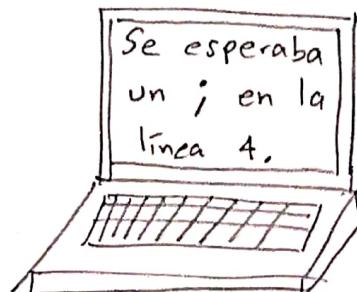
En esta estrategia, el analizador sintáctico realiza correcciones locales en la entrada restante cuando se encuentra un error. Por ejemplo, podría reemplazar una coma por un punto y coma, eliminar un punto y coma adicional o insertar un punto y coma que falta. La elección de la corrección local depende del diseñador del compilador. Sin embargo, debe hacerse con cuidado para evitar bucles infinitos, como sucedería si siempre se insertara algo en la entrada antes del símbolo actual.

3. Producciones de Error

Esta estrategia implica expandir la gramática del lenguaje con producciones que generen construcciones incorrectas comunes. Cuando se utiliza una producción de error, el analizador sintáctico puede generar mensajes de error adecuados para indicar la construcción incorrecta encontrada en la entrada. Es útil cuando se conocen los errores típicos que pueden ocurrir.



Puede
Ocasionalmente
bucles infinitos



Sintax
Error
X :c ...

4. Corrección Global

Esta estrategia idealmente minimiza los cambios necesarios al procesar una cadena de entrada incorrecta. Existen algoritmos que pueden encontrar una secuencia mínima de cambios para obtener una corrección global de menor costo.

Estos algoritmos buscan un árbol de análisis sintáctico para una cadena relacionada que minimice el número de inserciones, eliminaciones y modificaciones de componentes léxicos necesarios para transformar la entrada incorrecta en la entrada corregida. Sin embargo, la implementación de estos métodos suele ser costosa en términos de espacio y tiempo y, en la práctica, son más teóricos que prácticos.

Estas estrategias nos han proporcionado una comprensión más profunda de cómo los compiladores pueden enfrentar y recuperarse de los errores sintácticos en los programas. Cada una tiene sus ventajas y desventajas, y entenderlas nos ayudará a diseñar compiladores más robustos y eficientes en el futuro.

YACC / BISON

YACC es un acrónimo en inglés (Yet Another Compiler Compiler) que en español podemos traducir como "otro compilador de compiladores", es decir, podemos entender a YACC como una herramienta que nos trae la posibilidad de generar el código fuente de un compilador para algún nuevo lenguaje de programación.

Esta herramienta de software, además tiene un gran campo de utilidad en el diseño y desarrollo de compiladores y analizadores sintácticos en lenguajes de programación.

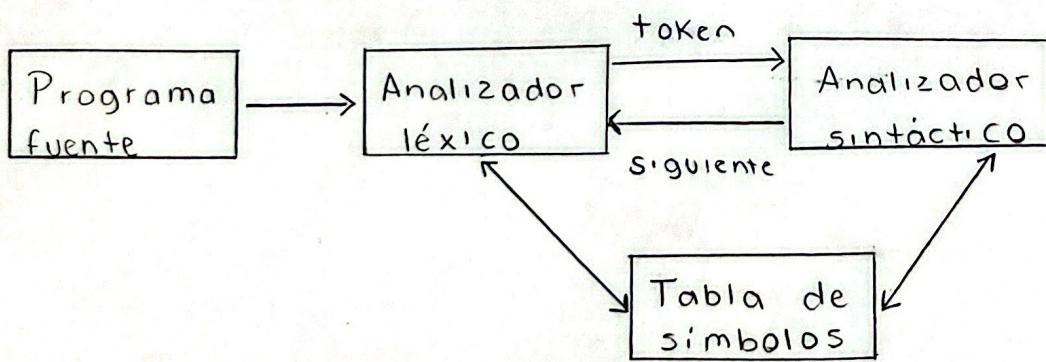
Genera un analizador sintáctico (en el proceso de compilación) este prueba que la estructura del código fuente se ajuste a la especificación sintáctica del

lenguaje en contexto) debido a que está basado en una gramática analítica escrita en una notación similar a la notación BNF. Este análisis sintáctico produce un árbol de análisis para seguir con su ingreso a otro proceso que sale del tópico o explicar, el análisis semántico.

De igual manera Bison requiere ser provisto de una gramática libre de contexto para su funcionamiento, de hecho este analizador sintáctico es completamente compatible con YACC.

A nivel de estructura se presentan tres bloques principales : reglas, procedimientos de usuario, reglas.

- Definiciones: Contiene código en C, se definen símbolos terminales y no terminales, entre otras directivas de Yacc. Está delimitado por %.{ y %.}.
- Reglas: contiene reglas de producción y acciones a ejecutar cuando el analizador sintáctico aplica una regla de producción para hacer una sustitución. Las reglas del lenguaje son escritas en una notación similar a BNF.
- Procedimientos de usuario: contiene código C que requiere el analizador sintáctico.



La ilustración muestra la relación en el proceso de compilación entre el analizador léxico y el sintáctico.

Referencias Bibliográficas

- Carlos, I. (2011, marzo 6). 004- Expresiones y Árboles Sintácticos. Introducción a la programación. <https://ici800a2011.wordpress.com/2011/03/06/004-expresiones-y-arboles-sintacticos/>
- Toledano, J. S. (2021, octubre 9). Cómo crear un árbol binario de expresiones - Yo, Toledano. Toledano.org. <https://yo.toledano.org/desarrollo/como-crear-un-arbol-binario-de-expresiones.html>
- Tyler MSFT. (s.f.) Operandos y expresiones. Microsoft. com. Recuperado el 7 de octubre de 2023, de <https://learn.microsoft/es-es/cpp/c-language/operands-and-expressions?view=msvc-170>
- Tyler MSFT. (s.f.) Precedencia y orden de evaluación. Microsoft. com. Recuperado el 7 de octubre de 2023, de <https://learn.microsoft.com/es-es/cpp/c-language/precedence-and-order-of-evaluation?view=msvc-170>
- CIDECAME. UAEH. (s.f.). 3. I. 1 Manejo de los Errores Sintácticos. Recuperado el 8 de octubre de 2023, de <http://cidacame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/311-manejo-de-los-errores-sintacticos.html>
- CIDECAME. UAEH. (s.f.). 3.I. 2 Estrategias de Recuperación de Errores. Recuperado el 8 de octubre de 2023, de <http://cidacame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/312-estrategias-de-recuperacion-de-errores.html>

- Departamento de Lenguas Y Sistemas Informáticos. (Cs.f.). Análisis Sintáctico. Analizadores descendentes. UNED. Recuperado 8 de octubre de 2023, de : <https://www.cartagena99.com/recursos/alumnos/apuntes/PDL-07-Tema%201-Analisis%20sintactico%20descendente.pdf>
- 3.5 Análisis sintáctico descendente. (Cs.f.). <http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/35-analisis-sintactico-descendente.html>
- Universidad Europea de Madrid. (Cs.f.). Analizadores sintácticos. LAUREATE INTERNATIONAL UNIVERSITIES. Recuperado 8 de octubre de 2023, de <https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2-M4-U3-T2.pdf>
- 3.4 Análisis sintáctico Ascendente. (Cs.f.). Edu.mx. Recuperado el 8 de octubre de 2023, de <http://cidecame.uaeh.mx/lcc/mapa/PROYECTO/libro32/34-analisis-sintactico-ascendente.html>
- [clases], P.B. [@pablobarenbaumclases5093]. (2020, octubre 8). PGC05.- I Análisis sintáctico ascendente. Youtube. <https://www.youtube.com/watch?v=5lak1bnWiBc>
- Departamento de Ingeniería Informática, I.T.S. (Cs.f.). Procedimientos de Lenguajes. Cartagena99. Recuperado el 8 de octubre de 2023, de <https://www.cartagena99.com/recursos/alumnos/apuntes/PDL-08-Tema%205-Analisis%20sintactico%20ascendente.pdf>

- Universidad de Guanajuato. (2022, julio 16). Clase digital 11. Análisis sintáctico: Análisis ascendente LR. Recursos Educativos Abiertos: Sistema Universitario de Multimodalidad Educativo (SUME) - Universidad de Guanajuato.
<https://blogs.ugto.mx/rear/clase-digital-11-analisis-sintactico-analisis-ascendente-ir/>
- Uhu.cs. (s.f.). Análisis sintáctico ascendente. Tema 4.
<http://www.uhu.es/francesco.moreno/gii-p1/docs/Tema4.pdf>
- Juan Fuente, A.A. (2006) Tablas de símbolos de procesadores de lenguaje. Google. Available at:
<https://chrome.google.com/webstore/detail/adobe-acrobat-pdf-edit-co/efaidnbmnnnibpcasjpcgkclefindmka>
(Accessed: 05 October 2023).
- Generación de la tabla de Símbolos (2020) Youtube.
Available at: <https://www.youtube.com/watch?v=y08ba5GxcUU> (Accessed: 05 October 2023).
- Aho, A., Lam, M., Sethi, R., & Ullman, J. (2006). Compilers: Principles, Techniques, and Tools. Pearson Education.



TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA

ANÁLISIS SINTÁCTICO.

Analizadores Sintácticos Haciendo Uso De Bison/Yacc y Flex

"Orgullosamente Lince"

Nombre de la asignatura:	LENGUAJES Y AUTOMÁTAS II
Carrera:	INGENIERÍA EN SISTEMAS COMPUTACIONALES
Equipo:	1
Autores:	ARROYO GÓMEZ JOSÉ ALFREDO (20030029) GARCÍA HERNÁNDEZ CESAR (20030853) GASCA PALACIO JESÚS FERNANDO (20030606) GONZÁLEZ MANCERA CHRISTIAN MANUEL (200300115)
Fecha Realización / Fecha Entrega:	03/10/2023 – 10/10/2023

1

INTRODUCCIÓN

En el apasionante mundo de la programación, los compiladores desempeñan un papel fundamental al traducir el código fuente escrito por los humanos en un lenguaje de alto nivel a un formato que la máquina puede entender y ejecutar. En esta práctica, nos aventuraremos en la creación de un mini compilador utilizando dos herramientas poderosas: Bison y Flex.

Nuestro objetivo principal será diseñar un compilador que pueda analizar y procesar listas, un concepto fundamental en la programación. Además, crearemos un compilador que pueda reconocer y analizar operaciones aritméticas simples.

Bison y Flex son herramientas ampliamente utilizadas en la construcción de compiladores y nos permitirán definir la gramática y analizar el código fuente de entrada de manera efectiva.

A lo largo de esta práctica, aprenderemos a definir reglas gramaticales utilizando Bison, que serán esenciales para reconocer y estructurar las listas. Flex, por otro lado, nos ayudará a escanear el código fuente y convertirlo en tokens que Bison utilizará para construir un árbol sintáctico.

El Programa 1 es una implementación de un analizador léxico y sintáctico que procesa una lista de números separados por comas. Este programa se basa en el lenguaje de programación C y utiliza herramientas como Bison y Flex para realizar el análisis. Su objetivo principal es analizar la entrada de texto en busca de números y mostrarlos en la salida estándar.

El Programa 2 es un evaluador de expresiones matemáticas tipo calculadora que puede realizar operaciones de suma, resta, multiplicación y división. Este programa también se basa en el lenguaje de programación C y utiliza Bison para la generación del analizador sintáctico. Su objetivo principal es evaluar expresiones matemáticas dadas como entrada y mostrar el resultado en la salida estándar.

César García Hernández



TECNOLÓGICO
NACIONAL DE MÉXICO®



OBJETIVOS DE LA PRÁCTICA

- Comprensión de la gramática
- Aprender a utilizar Yacc o Bison para generar analizadores sintácticos a partir de una especificación de gramática.
- Aprender a especificar una gramática para un lenguaje de programación simple o un lenguaje específico
- Utilizar Bison o Yacc para construir un árbol de sintaxis abstracta (AST) que represente la estructura de las sentencias del lenguaje.
- Realizar pruebas exhaustivas para asegurarse de que el analizador sintáctico funcione correctamente. Utilizar herramientas de depuración para resolver problemas.
- Documentar la especificación de la gramática y cualquier otro aspecto relevante del analizador sintáctico.
- Experimentar con diferentes gramáticas y ejemplos de entrada para comprender cómo funciona el análisis sintáctico en diferentes contextos.
- Comparación con Flex
- Preparar una presentación o informe que explique el proceso de diseño, implementación y prueba del analizador sintáctico, y que incluya ejemplos de uso.

MARCO TEÓRICO

Los generadores de código para analizadores sintácticos YACC y Bison son herramientas que facilitan la creación de analizadores sintácticos (también conocidos como parsers) para el procesamiento de lenguajes formales. Ambos generadores son ampliamente utilizados en la construcción de compiladores, intérpretes y otras aplicaciones que involucran análisis de gramáticas contextuales. A continuación, se proporciona un marco teórico de cómo funcionan estos generadores y cuáles son sus características y funciones.

YACC (Yet Another Compiler Compiler):

Entrada de YACC: YACC recibe como entrada una especificación de la gramática en forma de reglas de producción y acciones semánticas asociadas a esas reglas. Esta especificación se describe en un archivo de entrada con extensión ".y".

Generación del analizador sintáctico: YACC genera un analizador sintáctico en lenguaje C (o en otros lenguajes en algunas implementaciones) a partir de la especificación de la gramática. Este analizador se encarga de analizar la estructura sintáctica de un programa fuente.

Automatización del análisis sintáctico: El analizador sintáctico generado por YACC utiliza un algoritmo de análisis sintáctico descendente (parsing) para verificar si la entrada cumple con la gramática especificada. Si la entrada es sintácticamente válida, el analizador puede ejecutar acciones semánticas asociadas a las reglas de producción.

Acciones semánticas: Además de generar el analizador sintáctico, YACC permite especificar acciones semánticas que se ejecutan cuando se reconocen ciertas estructuras sintácticas. Estas acciones permiten construir árboles de sintaxis abstracta o realizar acciones específicas durante el análisis.



Bison:

Entrada de Bison: Al igual que YACC, Bison toma como entrada una especificación de la gramática en forma de reglas de producción y acciones semánticas. La especificación se define en un archivo de entrada con extensión ".y", y la sintaxis es muy similar a la de YACC.

Generación del analizador sintáctico: Bison genera un analizador sintáctico en lenguaje C (u otros lenguajes según la implementación) a partir de la especificación de la gramática. La calidad y la eficiencia del analizador generado suelen ser comparables a los de YACC.

Características avanzadas: Bison ha evolucionado para incluir características adicionales y mejoras con respecto a YACC. Por ejemplo, Bison admite la generación de analizadores sintácticos LALR(1), que son más eficientes en términos de análisis sintáctico y manejo de conflictos.

Características y funciones de YACC y Bison:

- **Generación automática:** Ambas herramientas automatizan la generación de analizadores sintácticos a partir de una especificación de gramática, lo que ahorra tiempo y esfuerzo en el desarrollo de compiladores y otros sistemas relacionados.
- **Gestión de conflictos:** Tanto YACC como Bison son capaces de manejar conflictos en la gramática, como conflictos shift-reduce y reduce-reduce, mediante mecanismos de resolución de conflictos configurables.
- **Personalización:** Permiten personalizar el comportamiento del analizador sintáctico a través de acciones semánticas, lo que facilita la construcción de árboles de sintaxis abstracta y la implementación de la semántica del lenguaje.
- **Extensibilidad:** Ambos generadores de código son extensibles, lo que significa que se pueden agregar características adicionales al analizador sintáctico generado, como la generación de código intermedio o la integración con otros componentes del compilador.

YACC y Bison son generadores de código que simplifican la creación de analizadores sintácticos para procesar lenguajes formales. Aunque tienen algunas diferencias, comparten características clave y funciones esenciales en el desarrollo de compiladores y sistemas relacionados.

MATERIALES, EQUIPOS Y RECURSOS EN GENERAL A UTILIZAR

- **Equipos de cómputo (Nuestros equipos son de gama media)**
 - Procesadores Intel Core i3-i5
 - Memoria RAM de 8 – 16 GB
 - Discos duros de estado sólido con almacenamiento desde 180 GB hasta 500 GB
- **Internet (Dependiendo del proveedor y paquete contratado)**
- **Navegador web (Google, Opera, Mozilla, Edge, etc.)**
- **Editores de texto de ofimática (Word)**
- **Visual Studio Code**
- **Windows 10/11**
- **WLS, Máquina virtual con Ubuntu o S.O Ubuntu 23.04 (En nuestro caso usamos un WLS; Ubuntu en Windows)**

Para ejecutar el Programa 1, es necesario tener instalados los siguientes recursos:

- **Bison**
- **Flex**
- **Un compilador de C (por ejemplo, GCC, G++)**

Para ejecutar el Programa 2, es necesario tener instalados los siguientes recursos:

- **Bison**
- **Flex**
- **Un compilador de C (por ejemplo, GCC, G++)**

Para ejecutar el Programa 3, es necesario tener instalados los siguientes recursos:

- **Yacc**
- **Flex**
- **Un compilador de C (por ejemplo, GCC, G++)**

César García Hernández

DESARROLLO

Instalación de Bison/Yacc, Flex y compilador de C

En Linux utilizar los comandos:

- **sudo apt update**
- **sudo apt-get install bison**
- **sudo apt-get install flex**
- **sudo apt-get install gcc**

O

- **sudo apt install bison flex gcc**



El Programa 1 se desarrolla siguiendo una gramática definida en el archivo de entrada. El análisis se realiza en dos etapas: el análisis léxico (lexing) y el análisis sintáctico (parsing). Durante el análisis, se buscan números y se muestran en la salida estándar.

Léxico:

```
%{  
    #include<stdio.h>  
    #include "y.tab.h"  
    void yyerror(char *);  
}  
%option noyywrap  
  
%%  
  
", "           { return COMMA; }  
[0-9]+         { yylval = atoi(yytext); return NUM; }  
[ \t]+          { /* Ignorar espacios en blanco */ }  
\n             { return EOL; }  
.              { fprintf(stderr, "Carácter no válido: %s\n", yytext); }  
  
%%
```

1. `'{` y `'}': Estas marcas definen secciones de código que serán copiadas directamente en el archivo de salida generado por Flex (un archivo `c` que contendrá el analizador léxico). El código entre `'{` y `'}` se coloca generalmente al principio del archivo para incluir declaraciones y definiciones necesarias.
2. `#include <stdio.h>`: Esta línea incluye la biblioteca estándar de entrada/salida de C, que es necesaria para las funciones `printf()` y `FILE` utilizadas en el código.
3. `int yylex(void);`: Esta línea declara la función `yylex()`. Esta función es generada por Flex y se utiliza para realizar el análisis léxico del texto de entrada.
4. `void yyerror(char *s);`: Esta línea declara la función `yyerror()`. Esta función se utiliza para manejar errores léxicos y sintácticos durante el análisis.
5. `%token NUM COMMA EOL`: Aquí se definen los tokens que el analizador léxico reconocerá en el texto de entrada. En este caso, se definen tres tokens: `NUM` (número), `COMMA` (coma), y `EOL` (fin de línea). Estos tokens son utilizados en las reglas de análisis léxico más adelante.
6. `%%`: Esta línea marca el final de la sección de definición de tokens y el comienzo de las reglas de análisis léxico y sintáctico.
7. `list: /* Lista vacía */`: Esta es una regla de producción de la gramática. Define cómo se pueden construir listas de elementos. En este caso, indica que una lista puede estar vacía o contener



elementos seguidos de un salto de línea (`EOL`).

8. `|`: Este símbolo se utiliza para separar opciones en las reglas de producción. Por ejemplo, la producción puede ser una lista vacía o una lista con elementos.
9. `list item EOL{ printf("Número: %d\n", \$2); }`: Esta regla de producción define cómo se construye una lista con un elemento. Cuando se encuentra un número (`NUM`) seguido de un salto de línea (`EOL`), se imprime ese número utilizando `printf()`.
10. `item: NUM`: Esta regla de producción define cómo se construye un elemento (`item`). Simplemente es un número (`NUM`).
11. `item COMMA NUM { printf("Número: %d\n", \$3); }`: Esta regla de producción define cómo se construye un elemento con coma. Cuando se encuentra un elemento (`item`) seguido de una coma (`COMMA`) y otro número (`NUM`), se imprime ese último número utilizando `printf()`.
12. `int main() { return(yparse()); }`: Aquí se define la función `main()`. Esta función es la entrada principal del programa. Llama a `yparse()`, que es una función generada por Flex para iniciar el análisis léxico y sintáctico. La función `yparse()` devolverá un valor que indica si el análisis tuvo éxito o no.
13. `void yyerror(char *s) { printf("\n%s\n", s); }`: Aquí se define la función `yyerror()`. Esta función se encarga de imprimir mensajes de error en caso de que ocurra un error durante el análisis léxico y sintáctico.

Parsing

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

%token NUM COMMA EOL
%%

list: /* Lista vacía */
    | list item EOL{ printf("Número: %d\n", $2); }
    ;

item: NUM
    | item COMMA NUM { printf("Número: %d\n", $3); }
    ;

%%
int main(){
```



```

    return(yyparse());
}
void yyerror(char *s){
    printf("\n%s\n", s);
}

```

1. ` `%{` y ` %}`: Estas marcas definen secciones de código que se copiarán directamente en el archivo de salida generado por Flex. Aquí se incluyen las bibliotecas necesarias (`#include<stdio.h>`) y se declara la función `yyerror()` que se utilizará para manejar errores léxicos y sintácticos.
2. `#include "y.tab.h"`: Esta línea incluye el archivo de encabezado "y.tab.h", que generalmente es generado por Bison y contiene definiciones de tokens utilizadas en el analizador sintáctico (parser).
3. `void yyerror(char *);`: Declara la función `yyerror()`, que se utilizará para manejar errores léxicos y sintácticos durante el análisis.
4. `option noyywrap`: Esta opción indica que no se generará automáticamente una función `yywrap()`. `yywrap()` a menudo se utiliza para el procesamiento de archivos múltiples, pero en este caso no es necesario.
5. `"," { return COMMA; }`: Esta regla de patrón reconoce una coma (",") y devuelve el token `COMMA`. Esto significa que cuando se encuentre una coma en el texto de entrada, se considerará como un token "COMMA".
6. ` [0-9]+ { yyval = atoi(yytext); return NUM; }`: Esta regla de patrón reconoce secuencias de uno o más dígitos (números) y las convierte en un valor entero usando `atoi()`. Luego, devuelve el token `NUM` junto con el valor numérico asignado a `yyval`. Esto significa que cuando se encuentre un número en el texto de entrada, se considerará como un token "NUM" y su valor numérico estará disponible a través de `yyval`.
7. `[\t]+ { /* Ignorar espacios en blanco */ }`: Esta regla de patrón reconoce secuencias de uno o más espacios en blanco o tabulaciones y las ignora. No se devuelve ningún token para los espacios en blanco o tabulaciones.
8. ` \n { return EOL; }`: Esta regla de patrón reconoce saltos de línea ("\\n") y devuelve el token `EOL`. Esto significa que cuando se encuentra un salto de línea en el texto de entrada, se considerará como un token "EOL".
9. `.`: Este patrón coincide con cualquier otro carácter que no haya sido coincidido por las reglas anteriores. Cuando se encuentra un carácter que no se ajusta a ninguna de las reglas anteriores, se imprime un mensaje de error en `stderr` indicando que el carácter no es válido.



```

Microsoft Windows [Versión 10.0.22621.2283]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\chris\OneDrive\Documentos\Bison\AnalisisNuemros>a.exe
1,3,5,4,7,8
Número: 3
Número: 5
Número: 4
Número: 7
Número: 8
Número: 1
xc
Carácter no válido: x
Carácter no válido: c

syntax error

C:\Users\chris\OneDrive\Documentos\Bison\AnalisisNuemros>

```

El Programa 2 se desarrolla utilizando Bison para definir la gramática de las expresiones matemáticas y realizar el análisis sintáctico. Durante la evaluación, se realizan operaciones matemáticas y se muestra el resultado en la salida estándar.

Parsing

```

%{
#include<stdio.h>
int yylex(void);
void yyerror(char *s);
%}
%token NUMBER EVALUAR

%start INICIO
%left '+' '-'
%left '*' '/'

%%
INICIO
: EVALUAR '(' Expr ')' ';' 
{
    printf("\nResultado=%d\n", $3);
    return 0;
}
;

Expr
: Expr '+' Expr
{
    $$ = $1 + $3;
}
| Expr '-' Expr
{

```



```

    $$ = $1 - $3;
}
| Expr '*' Expr
{
    $$ = $1 * $3;
}
| Expr '/' Expr
{
    $$ = $1 / $3;
}
| NUMBER{
    $$ = $1;
}
;

%%
int main(){
    return(yparse());
}
void yyerror(char *s){
    printf("\n%s\n", s);
}
int yywrap(){
    return 1;
}

```

1. `'{` y `'}': Estas marcas definen secciones de código que serán copiadas directamente en el archivo de salida generado por Bison (generalmente un archivo `.c`). Aquí se incluyen las bibliotecas necesarias (`#include <stdio.h>`) y se declaran las funciones `yylex()` y `yyerror()`, que se utilizan en el código generado.
2. `int yylex(void);`: Esta línea declara la función `yylex()`. En este contexto, `yylex()` se utiliza para obtener tokens del analizador léxico (Flex) para que el analizador sintáctico (Bison) pueda procesarlos.
3. `void yyerror(char *s);`: Esta línea declara la función `yyerror()`, que se utiliza para manejar errores sintácticos durante el análisis.
4. `%token NUMBER EVALUAR`: Aquí se definen los tokens que serán utilizados en el analizador léxico y reconocidos por el analizador sintáctico. En este caso, se definen dos tokens: `NUMBER` y `EVALUAR`, que se utilizarán para representar números y la palabra clave "EVALUAR", respectivamente.
5. `%start INICIO`: Define el símbolo inicial de la gramática, que en este caso es "INICIO". Esto indica desde dónde comenzará el análisis sintáctico.
6. `%left '+' '-': Define la precedencia de los operadores de suma (+) y resta (-) para resolver



ambigüedades en la gramática.

7. `%left " /` : Define la precedencia de los operadores de multiplicación (` `) y división (` /`) para resolver ambigüedades en la gramática.
8. `INICIO: EVALUAR '(' Expr ')' ';'` : Esta regla de producción define cómo se debe estructurar una entrada válida. Debe comenzar con la palabra clave "EVALUAR", seguida de un paréntesis izquierdo, seguida de una expresión (`Expr`), seguida de un paréntesis derecho y un punto y coma.
9. `'{ printf("\nResultado=%d\n", \$3); return 0; }` : Esta acción se ejecutará cuando se encuentre una entrada válida. Imprimirá el resultado de la expresión (`\$3`) y devolverá un valor de 0.
10. `Expr: ...` : Aquí se definen las reglas de producción para las expresiones matemáticas. Esto incluye cómo se pueden combinar números (`NUMBER`) y operadores (+, -, *, `/`) para formar expresiones más complejas. Cada regla define cómo se calcula el resultado de la expresión utilizando `\$1`, `\$2`, `\$3`, etc., para representar los valores calculados en las producciones anteriores.
11. `int main() { return(yyparse()); }` : Define la función `main()` que inicia el análisis sintáctico llamando a `yyparse()`. Devuelve un valor para indicar si el análisis fue exitoso o no.
12. `void yyerror(char *s) { printf("\n%s\n", s); }` : Define la función `yyerror()` que se utiliza para manejar errores sintácticos y muestra un mensaje de error en la consola.
13. `int yywrap() { return 1; }` : Define la función `yywrap()` que se utiliza para indicar el final de la entrada. En este caso, siempre devuelve 1 para indicar el final de la entrada.

Léxico

```
%{  
#include<stdio.h>  
#include "y.tab.h"  
void yyerror(char *);  
}  
  
%option noyywrap  
  
DIGIT    [0-9]  
NUM      {DIGIT}+("."{DIGIT}+)?  
  
/* Reglas gramaticales */  
%%  
{NUM}          { yyval=atoi(yytext); return NUMBER; }  
[-()/*;/]     { return *yytext; }  
"evaluar"     { return EVALUAR; }  
[[::blank:]] ;
```



.
%%

```
yyerror("Carácter inválido");
```

1. `%%` y `%%`: Estas marcas definen secciones de código que se copiarán directamente en el archivo de salida generado por Flex. En este caso, se incluyen las bibliotecas necesarias (`#include <stdio.h>`) y se declara la función `yyerror()`, que se utilizará para manejar errores léxicos y sintácticos.
2. `#include "y.tab.h"`: Esta línea incluye el archivo de encabezado "y.tab.h", que generalmente es generado por Bison y contiene definiciones de tokens y reglas de producción utilizadas en el analizador sintáctico (parser).
3. `void yyerror(char *);`: Declara la función `yyerror()`, que se utilizará para manejar errores léxicos y sintácticos durante el análisis.
4. `%option noyywrap`: Esta opción indica que no se generará automáticamente una función `yywrap()`. `yywrap()` a menudo se utiliza para el procesamiento de archivos múltiples, pero en este caso no es necesario.
 1. 5. `DIGIT [0-9]`: Esto define una regla llamada `DIGIT` que coincide con un solo dígito del 0 al 9.
 2. 6. `NUM {DIGIT}+("."{DIGIT}+)?`: Esto define una regla llamada `NUM` que coincide con números decimales, por ejemplo, "123.45". La parte entera se representa como `{DIGIT}+`, seguida opcionalmente por un punto (`.`) y una parte decimal que también se representa como `{DIGIT}+`.
 3. 7. `%%`: Esto marca el final de las definiciones y el comienzo de las reglas de acción.
 4. 8. `'{NUM} { yyval=atoi(yytext); return NUMBER; }`: Esta regla de acción se ejecutará cuando se encuentre un número decimal. La acción convierte el texto del número en un valor entero usando `atoi()` y lo almacena en `yyval`. Luego, se devuelve el token `NUMBER`.
 10. `[-()+;/] { return *yytext; }`: Esta regla de acción se ejecutará cuando se encuentren caracteres como `-`, `+`, `(`, `)`, ` `, `/`, o `;`. Estos caracteres se devuelven como tokens individuales.
 11. `'"evaluar" { return EVALUAR; }`: Esta regla de acción se ejecutará cuando se encuentre la palabra clave "evaluar". Devuelve el token `EVALUAR`.
 12. `'[[:blank:]] ;`: Esta regla de acción ignora espacios en blanco y caracteres de tabulación.
 5. 12. `::`: Esta regla de acción coincide con cualquier otro carácter que no haya sido coincidido por las reglas anteriores. Cuando se encuentra un carácter que no coincide con ninguna regla, se llama a la función `yyerror()` para indicar que el carácter es inválido.



```
Microsoft Windows [Versión 10.0.22621.2283]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\chris\OneDrive\Documentos\Bison>a.exe
evaluar(4+4);

Resultado=8

C:\Users\chris\OneDrive\Documentos\Bison>
```

Programa 3. Conversor de números binarios a decimales

Crearemos un analizador sintáctico para un lenguaje de programación muy simple que permita convertir números en base 2 a números en base decimal.

Definimos nuestro analizador sintáctico:

```
%{
/* Definition section */
#include<stdio.h>
#include<stdlib.h>
void yyerror(char *s);
%}
%token ZERO ONE

/* Rule Section */
%%
N: L {printf("\n%d", $$);}
L: L B {$$=$1*2+$2;}
| B {$$=$1;}
B:ZERO {$$=$1;}
|ONE {$$=$1;};
%%

//driver code
int main()
{
while(yyparse());
}

yyerror(char *s)
{
```



```
fprintf(stdout, "\n%s\n", s);
}
```

1. En la sección de definición simplemente se incluyen las librerías estándar de C y asimismo se define un método vacío para manejar los errores void yyerror(char *s) que tiene como entrada un puntero de tipo char.
2. Posteriormente se definen los Tokens ZERO ONE que se usarán en la gramática. Representan los dígitos binarios 0 y 1, respectivamente.
3. La sección %% es donde se definen las reglas de producción de la gramática. Las reglas especifican cómo se deben analizar las entradas y cómo se deben realizar las acciones semánticas.
4. N: L {printf("\n%d", \$\$);}: Esta regla define un símbolo no terminal N, que representa un número decimal. Cuando se reconoce un número, se imprime su valor decimal utilizando printf.
5. L: L B {\$\$=\$1*2+\$2;}: Esta regla define un símbolo no terminal L que representa una secuencia de dígitos binarios. La regla indica que L puede ser una secuencia de L seguida de un dígito binario B. La acción semántica calcula el valor decimal de la secuencia binaria.
6. L: B {\$\$=\$1;}: Esta regla define otra forma en que L puede representar un dígito binario directamente.
7. B: ZERO {\$\$=\$1;} y B: ONE {\$\$=\$1;}: Estas reglas definen el símbolo no terminal B, que representa un dígito binario. Dependiendo de si es un ZERO o un ONE, la acción semántica asigna el valor apropiado.
8. Driver code (Código principal): En la función main, se ejecuta un bucle que llama a yyparse() repetidamente. yyparse() es una función generada por YACC que inicia el análisis sintáctico y devuelve 0 cuando se alcanza el final de la entrada.
9. Función yyerror: Esta función se utiliza para manejar los errores sintácticos. En este caso, simplemente imprime el mensaje de error en la salida estándar.

Definimos nuestro analizador léxico:

```
%{
/* Definition section */
#include<stdio.h>
#include<stdlib.h>
#include"y.tab.h"
extern int yylval;
}

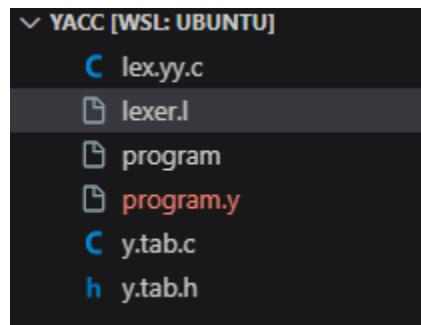
/* Rule Section */
%%
0 {yylval=0;return ZERO;}
1 {yylval=1;return ONE;}
```



```
[ \t] {;}
\n return 0;
. return yytext[0];
%%
```

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

1. En la sección de definición simplemente se incluyen las librerías estándar de C y también incluimos el archivo que se genera al usar el comando yacc y recibiendo como entrada nuestro parser, program.y. También definimos la variable yylval como extern para que pueda ser utilizada fuera de este archivo.
2. En la sección de reglas simplemente definimos los tokens ZERO, con el valor de 0 (cero) y ONE, con el valor de 1(uno).
3. Asimismo, definimos que las tabulaciones no generan una acción
4. Los saltos de línea retornan 0, es decir, no ocurre un error.
5. Y cualquier otro carácter simplemente no esta definido en el lenguaje.
6. Finalmente la función yywrap() es una función necesaria para que el analizador léxico funcione correctamente. Simplemente le indica al programa que ha llegado al final del archivo de entrada.



Aquí el parser manda un error sin embargo al ejecutar el programa es correcto.
Creemos que es una simple advertencia.



The image shows a code editor window with two tabs: "program.y 3 X" and "lexer.l". The "program.y" tab is active and displays the following Yacc grammar file:

```
1 %{
2 /* Definition section */
3 #include<stdio.h>
4 #include<stdlib.h>
5 void yyerror(char *s);
6 %}
7 %token ZERO ONE
8
9 /* Rule Section */
10 %%
11 N: L {printf("\n%d", $$); }
12 L: L B {$$=$1*2+$2; }
13 | B {$$=$1; }
14 B: ZERO {$$=$1; }
15 | ONE {$$=$1; }
16 %%
17 //driver code
18 int main()
19 {
20     while(yyparse());
21 }
22
23 yyerror(char *s) |
24 {
25     fprintf(stdout, "\n%s\n", s);
26 }
```





```
program.y 3    lexer.l    X

lexer.l
1 %{
2 /* Definition section */
3 #include<stdio.h>
4 #include<stdlib.h>
5 #include"y.tab.h"
6 extern int yylval;
7 %
8
9 /* Rule Section */
10 %%
11 0 {yylval=0;return ZERO;}
12 1 {yylval=1;return ONE;}
13
14 [\t]{;}
15 \n return 0;
16 . return yytext[0];
17 %%
18
19 →
20 int yywrap()
21 {
22 return 1;
23 }
```

Utilizamos la siguiente secuencia de comandos para compilar el parser, el lexer y el programa en si.

1. yacc -d program.y
2. flex lexer.l
3. gcc -o program y.tab.c lex.yy.c -lm

Posteriormente ejecutamos el programa:
./program

```
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS

● freddy024@DESKTOP-GH2FFV4:~/yacc$ ./program
0001

● 1freddy024@DESKTOP-GH2FFV4:~/yacc$ ./program
1010

● 10Freddy024@DESKTOP-GH2FFV4:~/yacc$ ./program
1101101

● 109freddy024@DESKTOP-GH2FFV4:~/yacc$ ./program
1101011011101

● 6877freddy024@DESKTOP-GH2FFV4:~/yacc$ ./program
11111111

○ 255freddy024@DESKTOP-GH2FFV4:~/yacc$
```

Podemos observar el correcto funcionamiento del programa

```
○ 255freddy024@DESKTOP-GH2FFV4:~/yacc$ ./program
01110a

14
syntax error

syntax error
```

Podemos observar que, si ingresamos valores erróneos, el analizador sintáctico manda un mensaje de error en el momento de encontrar tal error. Pero si antes de encontrar el error la cadena es válida y gramaticalmente es correcto, entonces, se ejecuta el programa hasta donde es correcto. Entonces reconoce el 01110 el cual es el número 14 en binario. Este tipo de manejo de errores es similar al Modo pánico.

BITÁCORA DE INCIDENCIAS

Fecha	Hora	Descripción de la incidencia	Solución
03/10/23	17:00	No conocíamos como instalar Bison y Flex para Windows	Buscamos documentación para poder instalar Bison y Flex.
03/10/23	19:00	No sabíamos que tipo de programa hacer con Bison	Buscamos un código de ejemplo, nos basamos en el para el código de la calculadora.
05/10/23	17:00	Tuvimos problemas con la gramática al generar desplazamientos innecesarios.	Leímos detenidamente la gramática descrita en nuestro programa y eliminamos las reglas innecesarias.
08/10/23	16:00	El analizador sintáctico marca un error en una línea, sin embargo, esto no genera conflicto con el ejecutable, o en algún proceso de compilación de estos.	Simplemente lo dejamos así debido a que no causa un error durante la compilación u ejecución del programa.

OBSERVACIONES

El Programa 1 ha logrado cumplir con éxito sus objetivos. Proporciona una solución efectiva para analizar listas de números en un formato específico. La combinación de Bison y Flex permite un análisis preciso de la entrada, y los resultados se muestran en la salida estándar de manera clara y legible.

El Programa 2 ha demostrado ser un evaluador de expresiones matemáticas eficiente y preciso. Puede procesar expresiones complejas y calcular el resultado de manera confiable. Esta herramienta es útil para aplicaciones que requieren la evaluación de fórmulas matemáticas en tiempo real.

El Programa 3 cuenta con un error a nivel del IDE de Visual Studio Code, sin embargo, el funcionamiento, compilación y ejecución es correcto.

César García Hernández



CONCLUSIONES

En esta práctica, hemos adentrado en el emocionante mundo de la construcción de compiladores utilizando Bison y Flex, dos herramientas poderosas en el ámbito de la programación. Hemos explorado dos escenarios interesantes: la creación de un mini compilador para listas y una pequeña calculadora.

A lo largo de esta experiencia, hemos aprendido cómo definir reglas gramaticales y utilizar Bison para construir analizadores sintácticos que pueden comprender y estructurar el código fuente de entrada. Flex, por su parte, nos ha permitido escanear y convertir el código fuente en tokens que Bison puede procesar.

En el caso del mini compilador para listas, hemos logrado desarrollar una herramienta capaz de reconocer estructuras de datos de lista. Esto nos ha brindado una valiosa comprensión de cómo los compiladores pueden ser diseñados para manipular datos complejos.

Por otro lado, en la creación de la pequeña calculadora, hemos construido una aplicación que evalúa expresiones matemáticas, incluyendo operaciones aritméticas básicas. Esto ha demostrado cómo las herramientas de construcción de compiladores pueden utilizarse para crear aplicaciones funcionales y útiles en el mundo real.

Finalmente, el programa para convertir números binarios en decimales es capaz de recibir cadenas en formato de 0's y 1's para poder mostrar su representación la cual es tamos acostumbrados los humanos. Asimismo, comprobamos el manejo de errores con el que cuentan Yacc y Bison. Esto podríamos aplicarlo o replicarlo en nuestros propios analizadores sintácticos.

Esta práctica nos ha proporcionado una base sólida en la construcción de compiladores, destacando la importancia de definir gramáticas precisas y utilizar herramientas como Bison y Flex para implementarlas. Estos conocimientos y habilidades son esenciales para cualquiera que aspire a comprender y crear compiladores y analizadores sintácticos, y son aplicables en una amplia variedad de campos de la informática y la programación.

En resumen, los Programas 1 y 2 son ejemplos de implementaciones de analizadores léxicos y sintácticos en C. Cada uno tiene sus objetivos específicos y utiliza herramientas como Bison y Flex para lograr su funcionalidad. Estos programas son útiles en el campo de procesamiento de lenguaje y evaluación de expresiones matemáticas.

César García Hernández

REFERENCIAS BIBLIOGRÁFIA

- *Bison for windows* (no date) GnuWin32. Available at: <https://gnuwin32.sourceforge.net/packages/bison.htm> (Accessed: 03 October 2023).
- *Win Flex-Bison* (2020) SourceForge. Available at: <https://sourceforge.net/projects/winflexbison/> (Accessed: 03 October 2023).
- *Construcción de una calculadora con flex y Bison - UAM* (no date) *Construcción de una calculadora con Flex y Bison*. Available at: http://arantxa.ii.uam.es/~mdlcruz/docencia/compiladores/2007_2008/construccion_calculadora_07_08.pdf (Accessed: 03 October 2023).
- T. T. (2019, mayo 9). YACC program for Binary to Decimal Conversion. GeeksforGeeks. <https://www.geeksforgeeks.org/yacc-program-for-binary-to-decimal-conversion/>

ANEXOS

Programas 1 y 2

- <https://drive.google.com/drive/folders/1WSqSF1c3nibqocrOo7MyZTZ3eGojrBj-?usp=sharing>

Programa 3

- https://drive.google.com/drive/folders/1ZBoZWjistUaqNB4oFESwCt-RbGXsPGfM?usp=drive_link

César García Hernández

Conclusiones

Los temas abordados en esta monografía desempeñan un papel crucial en la materia de Lenguajes y Automátas II, principalmente en la teoría de compiladores, diseño de lenguajes y el analizador sintáctico. Estos temas se interrelacionan para proporcionar una mejor compresión de cómo se diseñan, implementan y depuran los analizadores léxicos.

La precedencia de operadores es esencial para definir cómo se evalúan y agrupan las expresiones en un lenguaje de programación. Así para posteriormente, analizar sintácticamente ya sea, ascendente o descendente, el código fuente. Esta es la etapa central en el proceso de compilación pues verifica la estructura sintáctica del código fuente y la identifica para posteriormente indicar si pertenece a la gramática del lenguaje.

La administración de una tabla de símbolos es crucial para rastrear y gestionar la información sobre identificadores en un programa, como variables y funciones, lo que permite garantizar la coherencia de los símbolos, si son válidos o no. El manejo de errores sintácticos y las estrategias de recuperación son esenciales para proporcionar mensajes de error comprensibles y permitir que el análisis continúe incluso después de encontrar errores.

Por último, los generadores de código para analizadores sintácticos, como YACC y Bison, son herramientas muy poderosas que automatizan la generación de código para analizar la estructura sintáctica y ejecutar acciones correspondientes dependiendo de si son estructuras aceptadas o no. Estas herramientas son ampliamente utilizadas en la construcción de compiladores, lo que simplifica en gran medida el desarrollo de nuevos lenguajes de programación.

En conjunto, estos temas ofrecen una base sólida para comprender los principios teóricos y prácticos que subyacen en el diseño y la implementación de analizadores sintácticos y compiladores, y son esenciales para aquellos que buscan buenas formas de desarrollar y diseñar lenguajes de programación de forma rápida, práctica y precisa. En este caso, nosotros futuros ingenieros en sistemas computacionales.



TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA

Por qué ASML tiene el Monopolio Más COLOSAL de la Historia

“Orgullosamente Lince”

Nombre de la asignatura:	LENGUAJES Y AUTOMÁTAS II
Carrera:	INGENIERÍA EN SISTEMAS COMPUTACIONALES
Equipo:	1
Autores:	ARROYO GÓMEZ JOSÉ ALFREDO (20030029) GARCÍA HERNÁNDEZ CESAR (20030853) GASCA PALACIO JESÚS FERNANDO (20030606) GONZÁLEZ MANCERA CHRISTIAN MANUEL (200300115)
Fecha Realización / Fecha Entrega:	09/10/2023 – 09/10/2023

1

INTRODUCCIÓN

En esta tarea se realizó un proceso de análisis y síntesis sobre un video sobre el monopolio de la empresa ASML, la cual gobierna en el ámbito de la creación de chips debido sus máquinas capaces de realizar microchips 10,000 veces más pequeños que un pelo humano. En este documento presentamos el trabajo realizado para comprender esta información tan importante e interesante en el ámbito de la informática y computación.

2

OBJETIVOS DE LA PRÁCTICA

- Ejercitarse en el proceso de análisis y síntesis
- Desarrollar una documentación sobre un tema importante para el mundo de los microprocesadores
- Entender la importancia de los microchips
- Entender los conceptos más importantes en la creación de microchips

3

MARCO TEÓRICO

El monopolio de ASML en la industria de la fabricación de equipos de litografía y la creación de chips se debe a una combinación de innovación tecnológica constante, altos costos de entrada, relaciones comerciales sólidas, propiedad intelectual y efectos de red. Estos factores contribuyen a su dominio en el mercado y hacen que sea difícil que otros competidores puedan igualar su posición.

Algunos conceptos que no conocíamos y que no explican en el video los abordaremos aquí:

Litografía: Proceso de impresión que se utiliza para reproducir imágenes o texto sobre una superficie plana, generalmente papel, cartón, metal o piedra.

Litografía EUV: Técnica de impresión que utiliza luz ultravioleta extrema, con longitudes de onda mucho más cortas que las utilizadas en la litografía óptica convencional.



TECNOLÓGICO
NACIONAL DE MÉXICO®



MATERIALES, EQUIPOS Y RECURSOS EN GENERAL A UTILIZAR

- **Equipos de cómputo (Nuestros equipos son de gama media)**
 - Procesadores Intel Core i3-i5
 - Memoria RAM de 8 – 16 GB
 - Discos duros de estado sólido con almacenamiento desde 180 GB hasta 500 GB
- **Internet (Dependiendo del proveedor y paquete contratado)**
- **Navegador web (Google, Opera, Mozilla, Edge, etc.)**
- **Editores de texto de ofimática (Microsoft Word)**
- **YouTube**
- **Celular Gama Media**
- **Audífonos**

DESARROLLO

A continuación, se presentan algunos conceptos de importancia mencionados en el video:

En el mundo de la tecnología actualmente encontramos una gran competencia, pero esto no ha sido solo algo reciente, se viene viendo el comportamiento desde los inicios de la tecnología.

Con cada avance tecnológico, las empresas buscan la manera de utilizarlo para de esta forma poder sacar provecho.

En el presente video podemos observar que la marca ASML (Advanced Semiconductor Materials Lithography) actualmente tiene el monopolio en la creación y fabricación de chips, pero para llegar hasta este punto tuvo que recorrer un camino largo, lleno de dudas, crisis, incertidumbre y es aquí donde las empresas exitosas deben tomar acciones para poder afrontar lo que se presenta.

Las grandes empresas siempre se encuentran con dificultades, ya que el mercado es algo caótico, pero si se logran tomar las decisiones correctas en los momentos más difíciles, lo que resulta es un éxito rotundo.

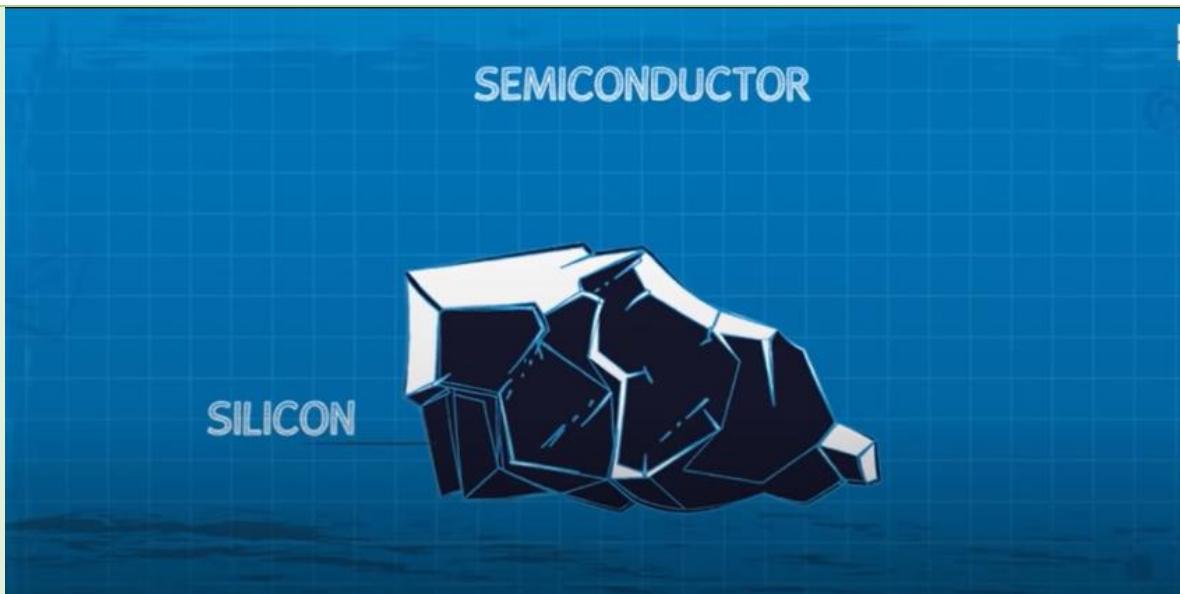
Podemos ver que en la historia de ASML, tenemos una competencia sobre la litografía con otras empresas como lo son Nikon y Canon las cuales lideraban el mercado, las cosas dieron un giro inesperado para los demás ya que ASM obtuve un avance que se llevó a cabo gracias a la inversión de una maquina twinscan.

Esta máquina lograba la medición e impresión en los chips para tomar una ventaja de 3 veces mayor a la de su competencia, esta diferencia fue suficiente para que la competencia para ASML quedara atrás.

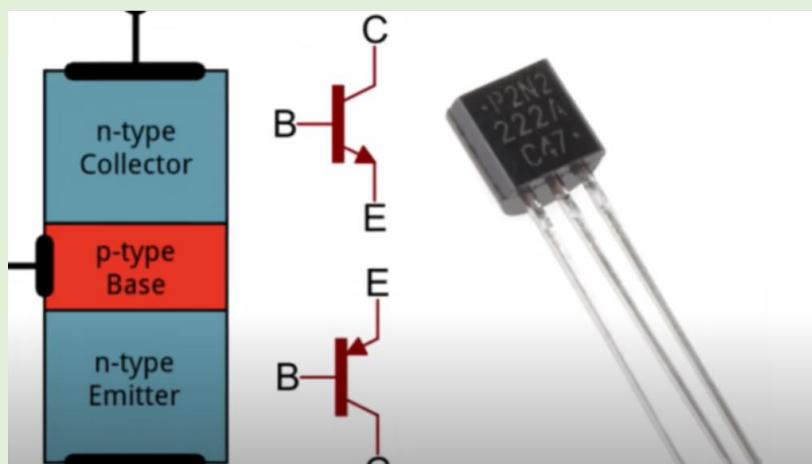
En el mundo moderno ya tenemos una amplia diversificación en cuestión de compañías ya que encontramos compañías que hacen software para el diseño de los chips, que diseñan el propio chip y algunas otras que se encargan de la venta de la maquinaria necesaria para la fabricación.

El único inconveniente es que ningún teléfono, sin importar la marca, solo podrá hacer teléfonos más poderosos si es posible que una máquina de ASML pueda llegar a fabricar transistores más pequeños





Semiconductor: Material que se comporta como conductor o no conductor de electricidad, lo que permite el registro de valores 1 o 0 según sea la presencia o ausencia de esta.



Transistor: Caminos por los cuales fluye la electricidad permitiendo o no el paso de esta, actúa como un interruptor. Hay que agregar que mientras más transistores tenga un chip, este tendrá mayor capacidad de procesamiento.

César García Hernández



Silicio: material de gran abundancia en la Tierra, su estructura atómica en adición de otros átomos permite que este se convierta en un material semiconductor.

Wafer: a partir de un lingote de silicio, se corta una parte de el en forma de disco. A el se añaden materiales en forma de capas que tienen un distinto uso.

- **Capa de fotorresistencia:** según sea la incidencia de luz, se endurece lo que permite que se graben patrones o caminos.
- **Fotomáscara:** cristal con patrones grabados en él, a través de este pasa luz que permite grabar en la capa de fotorresistencia.

En la parte final, a este wafer se le hace una limpieza de residuos para ser grabados y se agrega una capa metálica para permitir el paso de electricidad.

High NA: máquina planificada para salir al mercado en 2024, se tiene planeado que permitirá llevar a cabo la fabricación de chips con transistores de menor tamaño.

Algunas de las razones detrás de la dominancia de ASML en la industria de la fabricación de equipos de litografía:

1. **Tecnología de vanguardia:** ASML ha sido líder en el desarrollo de tecnologías de litografía avanzada, como la litografía ultravioleta extrema (EUV), que permite la producción de chips más pequeños y eficientes.
2. **Altos costos de entrada:** La fabricación de equipos de litografía avanzada es costosa y requiere una inversión significativa en investigación y desarrollo.
3. **Relaciones con fabricantes de chips:** ASML ha establecido relaciones sólidas y a largo plazo con los principales fabricantes de chips, como Intel, TSMC y Samsung.
4. **Economías de escala:** La producción de equipos de litografía a gran escala permite a ASML reducir los costos unitarios.
6. **Regulación y estándares de la industria:** La industria de los semiconductores a menudo está sujeta a regulaciones y estándares específicos. La posición de ASML como líder le permite influir en la creación de estándares y asegurarse de que sus tecnologías sean ampliamente adoptadas.
7. **Efectos de red:** La elección de ASML por parte de muchos fabricantes de chips crea un efecto de red, donde la compatibilidad y la estandarización se convierten en ventajas adicionales para la empresa.

6

BITÁCORA DE INCIDENCIAS

Fecha	Hora	Descripción de la incidencia	Solución
09/10/23	8:00	Tuvimos dudas relacionado al objetivo de la tarea a realizar, por lo que se fue a consultar con el profesor para aclarar dudas.	Simplemente le comentamos nuestras dudas al profesor y el amablemente nos comentó cuál era el objetivo de la tarea y como se debía realizar.

7

OBSERVACIONES

- Creemos que este tema es de vital importancia y no solo eso, es un tema actual, prácticamente de hace 2 semanas, que nos demuestra que la tecnología sigue en aumento. Entonces nos demuestra que debemos estar actualizados en todo momento.

8

CONCLUSIONES

ASML logró un monopolio destacado en la industria de la fabricación de chips y equipos de litografía, por su innovación tecnológica, barreras de entrada y relaciones sólidas con los fabricantes de chips, protección intelectual e influencia en estándares industriales y efectos de red.

Esta posición dominante plantea importantes cuestiones sobre la competencia en la industria de semiconductores y la necesidad de equilibrar la innovación y protección del mercado. La influencia de ASML en la fabricación de chips es un recordatorio de cómo las empresas pueden alcanzar un monopolio colosal en la era tecnológica.

9

REFERENCIAS BIBLIOGRÁFIA

- The Investing Corner [@theinvestingcorner]. (2023, septiembre 18). Por qué ASML tiene el Monopolio Más COLOSAL de la Historia. Youtube. https://www.youtube.com/watch?v=zAYCfw_syFc
- Litografía – HiSoUR Arte Cultura Historia. (s/f). Hisour.com. Recuperado el 9 de octubre de 2023, de <https://www.hisour.com/es/lithography-44045/>
- Torres, I. R. (2022, noviembre 5). Litografía EUV: ¿Qué es y por qué es el futuro? Profesional Review; Miguel Ángel Navas. <https://www.profesionalreview.com/2022/11/05/euv-que-es/>



TECNOLÓGICO
NACIONAL DE MÉXICO®



- https://www.youtube.com/watch?v=zAYCfw_syFc

César García Hernández

ENLACE AL VIDEO DE ESTA ACTIVIDAD:

https://drive.google.com/file/d/1_AwvTnlnPX8qLD0xNYBx8E2StVQIYh-5/view?usp=sharing

César García Hernández