

TECNOLÓGICO NACIONAL DE MÉXICO EN CELAYA

Asignatura: LENGUAJES & AUTOMÁTAS II

Docente: ISC. Ricardo González González

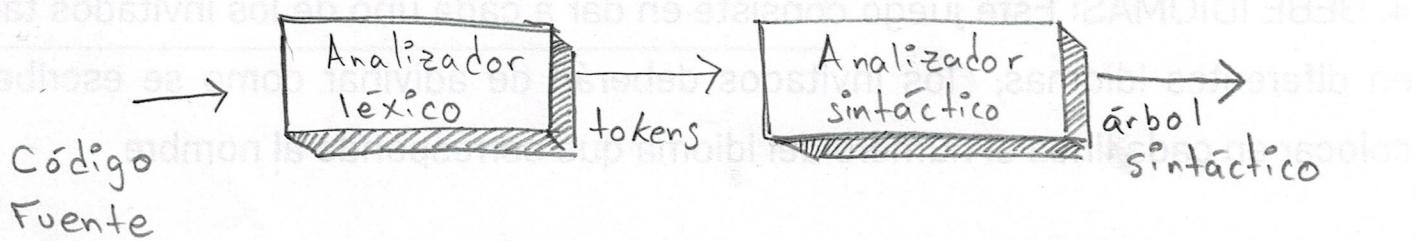
2023-09-18 LA-II-A

Actividad 4º Análisis Léxico &
Análisis Sintáctico

EQUIPO #1

INTEGRANTES:

- ARROYO GÓMEZ JOSÉ ALFREDO (20030029)
- GARCÍA HERNÁNDEZ CESAR (20030853)
- GASCA PALACIO JESÚS FERNANDO (20030606)
- GONZÁLEZ MANCERA CHRISTIAN MANUEL (20030115)





César García Hernández

DEPARTAMENTO DE SISTEMAS COMPUTACIONALES E INFORMÁTICA

ASUNTO: **SOLICITUD DE ACTIVIDADES**

Celaya, Guanajuato, 18 / septiembre / 202

LENGUAJES Y AUTÓMATAS II

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

ACTIVIDAD 4 (VALOR 44 PUNTOS)

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

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

2. ANÁLISIS LÉXICO.

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

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

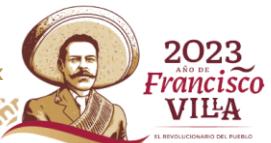
CONSIDERACIÓN :

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

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



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





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

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

3. ANÁLISIS SINTÁCTICO.

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

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

CONSIDERACIÓN :

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

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

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

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

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

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

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





¿QUÉ SE CALIFICARÁ ?

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

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

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

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

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 DIRETRICES DE LA GUÍA TUTORIAL.

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

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

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

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

OBSERVACIONES:

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

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

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



César García Hernández



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 , LI MÁS EL GRUPO (-A, -B, -C)
DOCUMENTO	: AI-ACTIVIDAD 1, P1-PRACTICA 1, R1-REPORTE 1, TI-TAREA 1, PG1-PROGRAMA, ETC. (CAMBIANDO EL NÚMERO CONSECUITIVO POR EL QUE CORRESPONDA)
[EQUIPO]	: NÚMERO DEL EQUIPO QUE CORRESPONDA SEGÚN INDICACIÓN DEL PROFESOR. [OPCIONAL]
NOCTROL	: SU NÚMERO DE CONTROL
APELLIDOS	: SUS APELLIDOS
NOMBRE	: SU NOMBRE
SEM	: EL PERÍODO SEMESTRAL EN CURSO: AGO-DIC

EJEMPLO :

SI EL TRABAJO SE SOLICITÓ EN EQUIPO.

2023-09-18_TNM_CELAYA_LAI-A_A4_EQUIPO_99_9999999_PEREZ_PEREZ_JUAN_AGO-DIC23.PDF

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

SI EL TRABAJO SE SOLICITÓ INDIVIDUALMENTE.

2023-09-18_TNM_CELAYA_LAI-A_A4_9999999_PEREZ_PEREZ_JUAN_AGO-DIC23.PDF



César García Hernández



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.



TECNOLOGICO NACIONAL
DE MÉXICO EN CELAYA

Asignatura: Lenguajes y Automátas II

Docente: Ricardo González González

2023 - 09 - 27

LAII - A

Actividad: Monografía
Análisis Léxico

Equipo #1

Integrantes

- Arroyo Gómez José Alfredo (20030029)
- García Hernández César (20030853)
- Gasca Palacio Jesús Fernando (20030606)
- González Mancera Christian Manuel (20030115)

Tabla de contenido

Índice de ilustraciones

Título: IMPORTANCIA Y APLICACIÓN DE LOS ANALIZADORES LÉXICOS Y SUS GENERADORES	1
INTRODUCCIÓN	1
GENERALIDADES	2
Contexto	2
Objetivos	2
DESARROLLO DEL TEMA	3
Fundamentos de analizadores léxicos	3
FUNCIONES DEL ANALIZADOR LÉXICO	6
CONCEPTOS BÁSICOS	6
SIMBOLOGÍA DE EXPRESIONES REGULARES	7
FUNCIONAMIENTO DEL ANALIZADOR LÉXICO	9
FORMAS DE IMPLEMENTAR UN ANALIZADOR LÉXICO	10
ERRORES LÉXICOS	11
Modo pánico	12
GENERADOR DE ANÁLISIS LÉXICO	13
Utilidades	13

Funcionamiento	15
Software generador de análisis léxico	16
Estructura de un generador de análisis léxico	17
Declaraciones	17
Reglas	18
Procedimientos de usuario	19
Métodos de generadores de análisis léxico	19
EJEMPLO DE APLICACIÓN FLEX (ANALIZADOR LÉXICO)	20
CONCLUSIÓN	30
REFERENCIAS BIBLIOGRÁFICAS	31

Índice de ilustraciones

Ilustración 1. Proceso del analizador léxico - - - - -	5
Ilustración 2. Relación analizador léxico-sintáctico y tabla de símbolos - - - - - - - - - - - - - - - - -	7
Ilustración 3. Funciones del analizador léxico - - - - -	8
Ilustración 4. Simbología de expresiones regulares - - - -	10
Ilustración 5. Ejemplo tokens, lexemas y patrones - - -	10
Ilustración 6. Entrada y salida de un generador de análisis léxico - - - - - - - - - - - - - - - - -	13
Ilustración 7. Tabla comparativa generador análisis léxico -	14
Ilustración 8. Estructura general de código de un generador de análisis léxico - - - - - - - - - - - - - - - - -	17
Ilustración 9. Bloque de código de declaraciones - - - -	18
Ilustración 10. Bloque de código de reglas - - - - -	18
Ilustración 11. Método main generado por defecto - - -	19

Título: IMPORTANCIA Y APLICACIÓN DE LOS ANALIZADORES LÉXICOS Y SUS GENERADORES

Introducción

El análisis léxico es una parte fundamental para la compilación de un programa. Ya sea un programa escrito en Java, C o Python, nosotros que usamos estos lenguajes de programación de alto nivel debemos reconocer que cada uno de estos pasan por el proceso del analizador léxico. Independientemente de si son lenguajes interpretados o compilados. Es de gran magnitud la importancia del análisis léxico que, por lo cual, nos compete realizar el siguiente documento donde establecemos los conocimientos fundamentales a la hora de tratar con esta fase de la compilación.

Queremos expresar los conceptos básicos necesarios para abrir paso a nuevos conceptos más avanzados en este tema, como el funcionamiento e implementación del analizador léxico, pero más que nada, este documento se centrará un poco más en los generadores de analizadores léxicos.

En este presente trabajo, presentamos los resultados de la investigación y práctica sobre los analizadores léxicos y ejemplos prácticos para reforzar el conocimiento adquirido, así como ilustraciones que acompañan a los diferentes conceptos. Esperamos que la información, prácticas, diagramas y los ejemplos estén acorde al nivel esperado y sean congruentes.

Generalidades

Contexto

La materia de Lenguajes y Autómatas II nos permite es estudio de las partes básicas, fundamentales y concretas de los sistemas informáticos como lo son las computadoras. Es por ello por lo que debemos estudiar temas como: El diseño del lenguaje, las fases de la compilación: análisis léxico, sintáctico, semántico, código intermedio y optimización. Todo esto con el fin de ser capaces de, por nuestra propia cuenta, poder hacer un análisis y síntesis, así como el compilador, para obtener lo que más nos interesa, es decir, el conocimiento y funcionamiento de las computadoras en un nivel más concreto. En este caso en específico, el análisis léxico.

Objetivos

- Entender y aplicar los conceptos básicos del análisis léxico
- Relacionar el análisis léxico con el análisis sintáctico y la tabla de símbolos
- Incrementar nuestro vocabulario haciendo uso de la terminología correcta en el ámbito del análisis léxico
- Comprender la importancia del análisis léxico en las fases de compilación
- Entender y aplicar los conceptos de generador de analizadores léxicos
- Hacer prácticas y ejercicios haciendo uso de generadores de analizadores léxicos

Desarrollo del tema

Antes de empezar a utilizar un generador de analizadores léxicos es importante conocer su funcionamiento, y antes de eso, debemos conocer el funcionamiento del analizador sintáctico y sus características. Empecemos por:

Fundamentos de analizadores léxicos

¿Qué es un analizador léxico? Esencialmente, es una de las fases que realiza el compilador, es más, es la primera fase, por lo tanto, da pie a las demás. Debemos de entenderlo como el principio del proceso de compilación posteriormente de recibir el código fuente.

¿Qué tareas realiza el analizador léxico?

- ▷ Recibir como entrada el código fuente (lectura).
- ▷ Dar como salida tokens

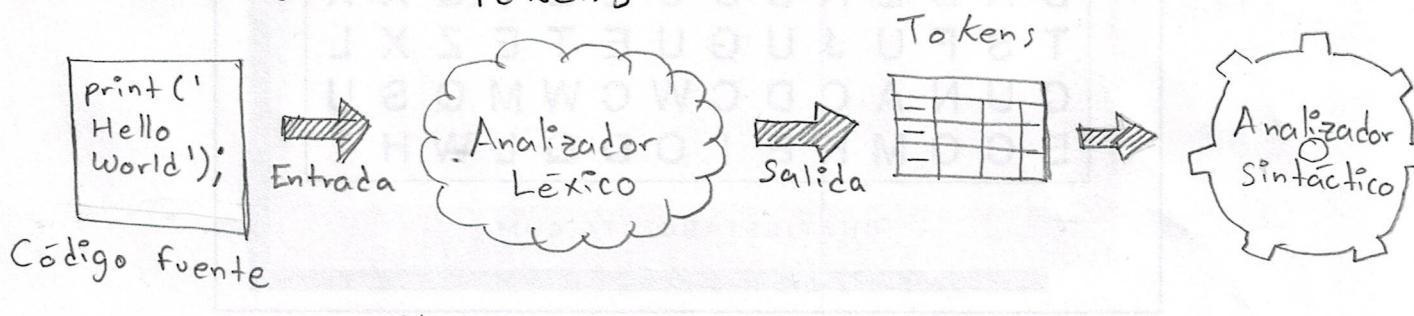


Ilustración I. Proceso análisis léxico

Ahora que hablamos de tokens, formalicemos.

¿Qué es un token y para qué sirve? En general podemos decir que un token es un identificador que puede ser asociado a un valor. En el caso para los analizadores léxicos, un token es un conjunto de símbolos reconocidos por el mismo analizador que forman las instrucciones del programa o código fuente.

Los tokens son enviados al analizador léxico en una estructura asociativa del tipo:

Token - Valor atributo

El valor del atributo o atributo es un valor numérico.

El proceso que realiza el analizador léxico: leer caracteres y enviar tokens lo hace bajo demanda, es decir, lo hace secuencialmente, carácter por carácter, en base a lo que pida el analizador sintáctico. Durante este proceso, si el token reconocido recae en la categoría de identificador, entonces, el analizador léxico envía su información a la tabla de símbolos.

¿Qué es la tabla de símbolos?

Es una estructura que almacena la información asociada con los tokens reconocidos por el analizador léxico. La información que almacena puede ser:

- Identificador
- Dirección de memoria
- Tipo de dato
- Tamaño o dimensión en bytes
- Entre otras

¿A nivel programación como se puede implementar una tabla de símbolos?

Con estructuras de datos del tipo clave-valor:

- Tabla hash
- Diccionarios
- Matrices asociativas
- JSON

```
Carros = [ { "marca": "Nissan",
              "color": "rojo" },
            { "marca": "BMW",
              "color": "blanco" },
            { "marca": "Chevrolet",
              "color": "negro" } ]
```

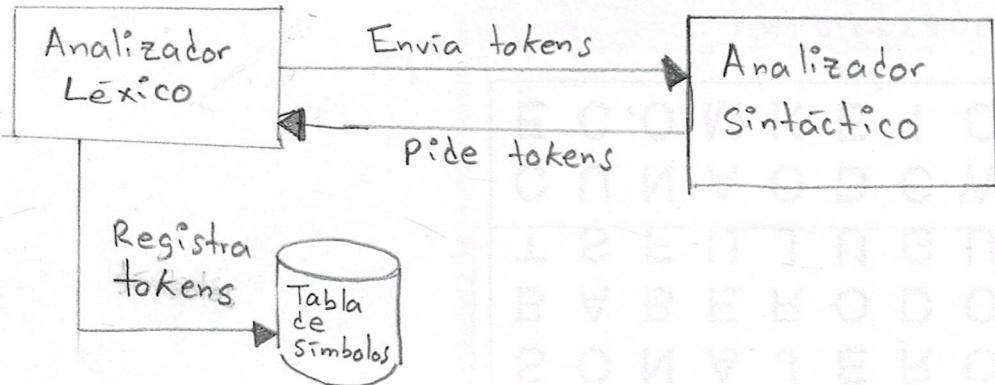


Ilustración 2. Relación analizador léxico-sintáctico y tabla de símbolos

Es de importancia agregar que el analizador léxico también cuenta con un sistema de gestión de errores. ¿Cómo funciona? Básicamente, si detecta un conjunto de caracteres, patrones o símbolos inválidos, o, en otras palabras, que no se reconocen en el analizador léxico, entonces, tenemos varias opciones:

- No hacer nada
- Indicar e informar el tipo de error → Así el programador pueda repararlo.
- Realizar un proceso de resolución de errores automático:
 - Desde la línea y columna donde se encontró el error o incongruencia: borrar, eliminar, modificar, depurar o reconstruir la cadena, hasta reconocer un símbolo o cadena válida y aceptada por el lenguaje.

Funciones del analizador léxico



Ilustración 3. Funciones del analizador léxico

Conceptos básicos

• **Token:** también llamados componentes léxicos, estos son conformados por una serie de símbolos y/o caracteres reconocidos por el analizador.

Representan a los símbolos terminales de la gramática
Pueden ser:

- Palabras reservadas
- Identificadores
- Operadores
- Constantes numéricas o de caracteres
- Caracteres especiales
- Comentarios
- Blancos
- Fin de entrada

• **Lexema:** conjunto de caracteres agrupados asociados a un token.

→ Los tokens pueden tener 1 o más lexemas.

▷ Identificadores: Relación 1 a muchos. (1 token: muchos lexemas).

▷ Palabras reservadas: Relación 1 a 1. (1 token: 1 lexema).

→ Son la representación de las operaciones tal y como las escribimos en el lenguaje de programación.

• **Patrón:** es el diseño, forma, estructura y/o relación entre los caracteres para formar un lexema. Generalmente representados por expresiones regulares.

• **Expresión regular:** Patrón a seguir para un determinado enunciado o lexema.

→ Ejemplos:

▷ Un dígito del 0 al 9 o una letra de la 'a' a la 'z'.
 $[0-9] \mid [a-z]$

▷ 3 minúsculas, un guión bajo y un dígito.
 $([a-z])^3 _ [0-9]$

▷ Al menos un símbolo de dólar
 $\$+$

Simbología de las expresiones regulares

Símbolo	Nombre	Descripción/Función	Exp. Regular	Ejemplos
.	Punto	Cualquier carácter menos salto de línea.	.	a, b, z, w, ...
*	Asterisco	Cero o más repeticiones.	$[a-z]^*$	E, abcd, aaad, perro, rata
+	Signo más	Una o más repeticiones.	$[a-z]^+$	abc, gato, s, a
^	Caret	El inicio de una línea o cadena.	\wedge Hola	Hola mundo Hola Hola, amigo

Símbolo	Nombre	Descripción / Función	Exp. Regular	Ejemplos
\$	Dolár	El final de una línea o cadena.	bien \$	bien muy bien te irá bien
[]	Corchetes	Conjunto de caracteres válidos. [aeiou]	a e i o u	
[^]	Corchetes con caret	Conjunto de caracteres no válidos.	[^019]	" Todo menos dígitos" a, -, \$, L, z
	Barra vertical	Operación OR	a b	a b
()	Paréntesis	Aplicar cuantificadores	(abc) ² (abc) ⁺ (abc)*	abcabc abc, abcabc, ... 3, abc, ...
\	Barra invertida	Escapar metacaracter	\n .	"Salto de línea" • /
?	Signo Interrogación	Opcional	a b (z)?	ab, abz //

Ilustración 4. Símbología de expresiones regulares

Tokens	Lexemas	Patrones
IF/condición	if	if
Adición/suma	+	\+
Identificador	\$, -abc, a, b, a1,	(\$ - [a-z])+ (\$ - [a-z] 0-9)*
Cadena	"abc", "01", "",	\" ([a-z] [0-9])* \"
Valor booleano	true, false	true false
Número	0, 123, 23.48,	[0-9]+ (\.[0-9]+)?

Ilustración 5. Tokens, lexemas y patrones

Funcionamiento del analizador léxico

1. El analizador léxico abre y lee el contenido del archivo fuente.
2. Se envía carácter por carácter bajo demanda al analizador sintáctico.
3. Si encontramos caracteres en blanco los descartamos.
4. Una vez encontramos un carácter que pueda ser identificado como un token, leemos ese carácter y retornamos su respectivo token.
5. El siguiente carácter encontrado posteriormente, si es el caso, debemos verificar si pertenece a alguno de los patrones previamente definidos por nuestras expresiones regulares.
6. Seguimos recorriendo la cadena del código fuente mientras no existan errores y todos los caracteres ingresados pertenezcan al lenguaje y los patrones identificados sean correctos.

Podemos representar nuestros patrones o expresiones regulares por medio de los autómatas finitos.

Autómata finito: programa que permite determinar si una cadena pertenece, o no, a un lenguaje con un alfabeto definido.

Definición formal

AF = 5-tupla: $A = (Q, \Sigma, \delta, q_0, F)$.

Q = conjunto de estados

q_0 = Estado inicial

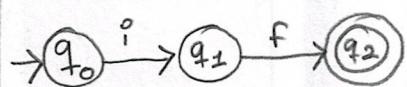
Σ = alfabeto de entrada

F = Conjunto de estados de aceptación/finales.

δ = funciones de transición

Ejemplos autómatas:

Reconocer "if"



$$Q = \{q_0, q_1, q_2\}$$

$$\Sigma = \{i, f\}$$

$$F = \{q_2\}$$

$$\delta = \delta(q_0, i) = q_1$$

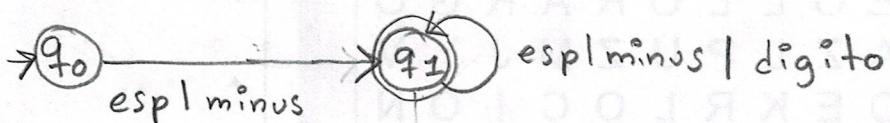
$$(q_1, f) = q_2$$

→ Solo acepta la cadena "i, f".

Reconocer un identificador

$$(\$ | - | [a-z])^+ (\$ | - | [a-z] | [0-9])^*$$

dígito = [0-9], minus = [a-z], esp = \\$ | -



$$Q = \{q_0, q_1\}$$

$$\Sigma = \{0, ..., 9, a, ..., z, -, \$\}$$

$$F = q_1$$

$$\delta = \delta(q_0, \text{esp}) = q_1$$

$$(q_0, \text{minus}) = q_1$$

$$(q_1, \text{esp}) = q_1$$

$$(q_1, \text{minus}) = q_1$$

$$(q_1, \text{dígito}) = q_1$$

Acepta:	✓
- mivar	
var	
v01-\$riable	

No acepta	X
01var	
@var	
234	

Formas de implementar un analizador léxico

- Generador de analizador léxico
 - + Fácil y práctico
 - Ineficiente y difícil de mantener
- Lenguaje de Alto Nivel
 - + Eficiente y compacto
 - Se realiza desde cero
- Lenguaje de Bajo Nivel
 - + Mucho más eficiente y compacto
 - Más difícil de implementar

Erros léxicos

Cuando se identifica un carácter no válido o algún patrón que no encaje con lo establecido en la tabla de símbolos, provoca un error de compilación.

Los posibles errores detectables son:

- Nombres de identificadores escritos incorrectamente
- Números incorrectos
- Palabras reservadas escritas incorrectamente
- Caracteres no válidos

Para implementar el manejo de errores del analizador léxico nos podemos valer del uso de una cola o pila.

Una posible estrategia para el manejo de errores es el siguiente:

- Si ocurre un error instantáneamente después de pasar por un estado de aceptación, entonces, realizamos la instrucción del estado de aceptación y el resto de los caracteres que contienen el error se retornan a la entrada.

Posteriormente nos movemos al estado inicial y a partir de ahí reconocemos el siguiente token.

- Si el error ocurre antes de entrar a un estado de aceptación entonces descartamos el carácter inválido y leemos el siguiente.

Modo pánico

Consiste en borrar caracteres inválidos hasta obtener un patrón, cadena o carácter válido. Para así formar el token.

1. Identificar el error y en qué estado sucedió
2. Ignorar, borrar, añadir o remplazar los caracteres inválidos
3. Obtener el siguiente token

Este modo es más utilizado cuando el lexema puede ser corregido con una sola acción, es decir, borrar, eliminar o añadir.

GENERADOR DE ANÁLISIS LÉXICO

También conocido como scanner o lexer, es posible afirmar que un generador de análisis léxico es una herramienta en formato de programa informático que cuenta con la capacidad de identificar y categorizar las partes más pequeñas de una cadena pudiendo ser números, símbolos o elementos de entrada (el concepto de token, ya mencionado durante el desarrollo de esta monografía).

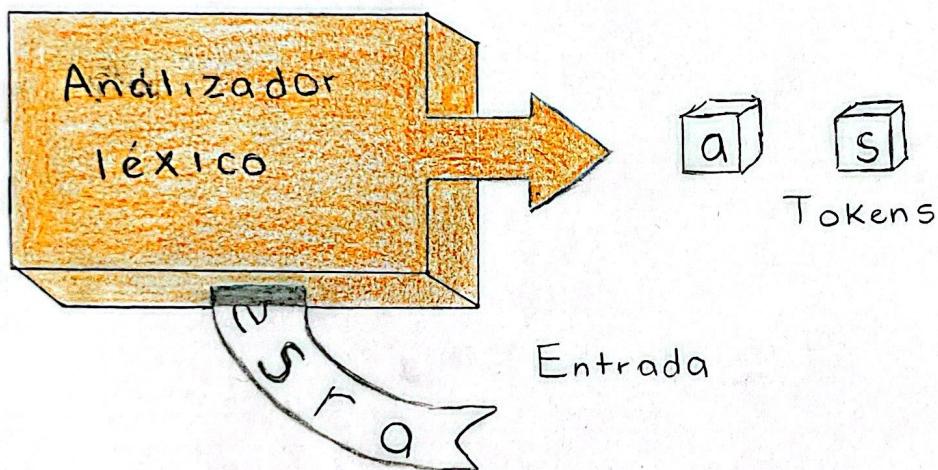


Ilustración b. Entrada y salida de un generador de análisis léxico, ilustrado por Hinostroza, T.

Utilidades

Este tipo de herramientas son de particular utilidad para la creación de programas que llevan a cabo procesamiento de texto.

cuales podemos encontrar algunos como:

- Lex
- Flex
- JT Lex

Se podría hacer mención de cada uno de ellos, pero resulta más práctico que sean expuestos mediante una tabla comparativa entre algunos de estos generadores, más específicamente los anteriormente mencionados.

Característica	Lex	Flex	JT Lex
Lenguaje de programación	C	C	Java
Desarrollador	AT & T Bell Laboratories	Vern Paxson	Gerwin Klein
Entrada	Archivo de texto	Archivo de texto	Archivo de texto
Portabilidad	Buena	Buena	Buena
¿Admite expresiones regulares?	Sí	Sí	Sí
¿Desarrollo constante?	No	Sí	Sí
Integración con analizador sintáctico.	Utilizado comúnmente con Yacc.	Utilizado comúnmente con Bison o Yacc.	Puede integrarse JavaCup.
Ecosistema	Limitado debido a su antigüedad	Amplio debido a su popularidad	Amplio.
Adicional	Generador de análisis léxico	Versión moderna de Lex	Alternativa a Flex en Java

Ilustración 7. Tabla comparativa generador análisis léxico

En conclusión, Lex es el generador de análisis léxico original y del cuál se comenzaron a basar distintas herramientas modernas. Flex tiene capacidad para ser más rápido si se utilizan expresiones regulares complejas, siendo más robusto para llevar a cabo proyectos de mayor complejidad.

- La construcción de compiladores o interpretes de lenguajes de programación, siendo responsable de reconocer elementos como identificadores, palabras reservadas y operadores.
- Crear software para la edición de texto como un editor de código fuente. Un analizador léxico puede identificar fallos y ofrecer recomendaciones para corregir estos.
- Hace apoyo al análisis sintáctico ya que este no tiene que preocuparse por llevar a cabo la revisión léxica, haciendo posible centrarse en comprender la estructura gramatical de la cadena de entrada brindada.

Funcionamiento

El generador de análisis léxico funciona mediante una serie de pasos que a continuación se describirán:

- 1: El primer paso es examinar la cadena de entrada carácter por carácter.
- 2: Utilizando reglas previamente establecidas se reconocen patrones que tengan relación con los tokens. Es de resaltar que estas reglas se expresan usualmente (o en su mayoría de ocasiones) como una expresión regular.
- 3: Cuando un patrón es reconocido y validado, se crea un token que suele llevar el tipo y valor del token, aunque dependerá de que se requiera.
- 4: Si existen caracteres que no coincidan con los patrones, el generador de análisis léxico crea un mensaje de error, posteriormente intenta seguir con el análisis.
- 5: El análisis sigue y se crea un flujo de tokens que representa la estructura léxica del código fuente.

Entonces, de forma general podemos establecer tres pasos esenciales:

- Definir
- Reglas
- Rutinas

Software generador de análisis léxico.

Ya hemos mencionado sobre el cómo actúa un generador de análisis léxico, sus utilidades y la manera en que opera, pero hay que saltar una pregunta que aún no se ha mencionado: ¿por qué usarlo? ¿qué beneficios conlleva?

Primordialmente supone gran facilidad en cuestiones de desarrollo de interpretes y compiladores, ya que al ser un generador su sólida será código fuente optimizado que supondrá ahorro de tiempo y recursos no solo a nivel de procesador, sino esfuerzo humano, de igual manera reduce abruptamente ser propenso a errores.

Facilita la mantenibilidad al mantener rigurosamente un código estructurado y bien organizado, haciendo posible identificar y resolver problemas ocasionados por errores, así mismo, permite tener un mejor panorama sobre implementaciones a futuro.

Separa responsabilidades, es decir, mantiene énfasis a nivel léxico siendo el encargado de reconocer posibles errores que se presenten evitando que esta tarea se lleve a cabo en el analizador sintáctico haciendo que éste únicamente se enfoque en la estructura del código y optimizando el tiempo de rendimiento.

Todas estas ventajas son provistas por software generador de análisis léxico, que llevan a cabo el paso de una expresión regular a un autómata finito, entre los

El que supone un menor rendimiento en tiempo es JTLex, pero eso no quiere decir que deba ser descartado su uso pues este ofrece una documentación extensa y una alta portabilidad al no requerir de bibliotecas externas.

Nos encontramos ante tres herramientas que sin duda han dejado huella, el uso de cada una de ellas dependerá más sobre los requisitos a cumplir y las preferencias del usuario.

Estructura de un generador de análisis léxico

Previamente hemos hecho mención de 3 pasos esenciales de un generador de análisis léxico: definir, reglas, sentencias.

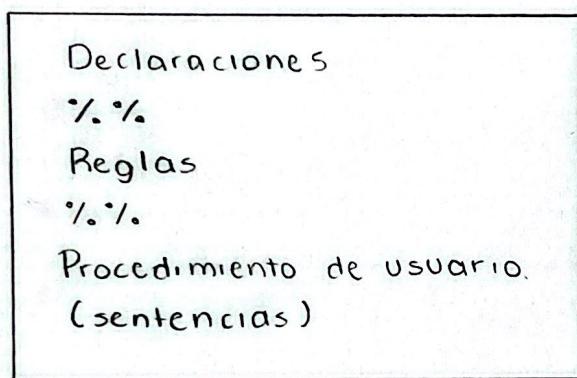


Ilustración 8. Estructura general de código de un generador de análisis léxico.

Esta estructura suele ser vista de forma general de un generador de análisis léxico, varía un poco según el software, pero nos permite tener un amplio panorama describiendo el funcionamiento.

Declaraciones

En este apartado es posible llevar a cabo la declaración de variables globales, agregar las bibliotecas

necesarias o incluso nombrar expresiones regulares a utilizar, viéndose de la siguiente manera.

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
int n1,np,nw  
entero [0-9]+  
%}
```

Ilustración 9. Se muestra el bloque de código de declaraciones.

Reglas

En este apartado se lleva a cabo la definición de reglas que dan paso al reconocimiento y procesamiento de los tokens de entrada, viéndose de la siguiente manera:

```
exp.reg → {acciones en C}  
[a-zA-Z_][a-zA-Z0-9_]* printf ("ID: %s", yytext); }
```

Ilustración 10. Se muestra el bloque de código de reglas.

Una regla existente por defecto es la siguiente:

```
{ECHO;}
```

Que consiste en que si se introduce algo que no sea reconocido lo va a imprimir tal cual, lo cual es posible mediante la instrucción.

```
. { }
```

De esta manera en lugar de mostrar lo que no se reconoció, mostrará un " ".

Procedimientos de usuario

Es posible escribir código en lenguaje C por ejemplo, para poder realizar procedimientos que el usuario requiera.

Por defecto, se agrega un método main con el siguiente código:

```
int main()
{
    yyin = stdin;
    yylex();
}
```

Ilustración 11. Método main generado por defecto.

Métodos de generadores de análisis léxico

Nos brindan funcionalidades, entre las cuales existen:

- `yylex()`: llama al analizador léxico.
- `yymore()`: añade el `yytext` actual al siguiente recorrido.
- `yyless(n)`: regresa `n` caracteres (últimos) a la cadena de ingreso
- `yyerror()`: se llama cuando un error léxico es encontrado.
- `yywrap()`: señala el final de la cadena de entrada y así controlar el comportamiento del analizador léxico.

TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE CELAYA

EJEMPLOS DE APLICACIÓN FLEX (ANALIZADOR LÉXICO). – Desarrollo de un programa que sirva de ejemplo del funcionamiento del analizador léxico

Nombre de la asignatura:	LENGUAJES Y AUTOMÁTAS II
Carrera:	INGENIERÍA EN SISTEMAS COMPUTACIONALES
Equipo:	3
Autores:	ARROYO GÓMEZ JOSÉ ALFREDO (20030029) GARCÍA HERNÁNDEZ CESAR (20030853) GASCA PALACIO JESÚS FERNANDO (20030606) GONZÁLEZ MANCERA CHRISTIAN MANUEL (200300115)
Fecha Realización / Fecha Entrega:	31 DE MARZO DEL 2023 / 18 DE ABRIL DEL 2023

1	INTRODUCCIÓN
En la presente práctica se muestra el proceso de realización de un programa que ejemplifica el funcionamiento de los analizadores léxicos usando generadores de los mismo, en este caso, flex.	

2	OBJETIVOS DE LA PRÁCTICA
<ul style="list-style-type: none">• Construir un analizador léxico simple• Generar una lista de tokens• Visualizar y comprender el proceso de manejo de errores• Documentar el proceso para realizar un analizador léxico con flex• Comprender la importancia de herramientas para generar analizadores léxicos	

3	MARCO TEÓRICO
<p>Vamos a agregar un pequeño marco teórico sobre flex ya que, en la actividad principal, el desarrollo de las monografías, ya hemos descrito algunos aspectos importantes de este generador de analizadores léxicos.</p> <p>Flex, abreviatura de "Fast Lexical Analyzer Generator," es una herramienta de software ampliamente utilizada en la construcción de analizadores léxicos para procesadores de lenguaje y compiladores. Su función principal es tomar una especificación de reglas léxicas escritas en un formato simple y generar automáticamente un analizador léxico eficiente en C o C++. Este analizador léxico es capaz de reconocer y clasificar tokens en un flujo de entrada, como palabras clave, identificadores, números y símbolos, lo que es esencial en la interpretación y compilación de lenguajes de programación. Flex simplifica el proceso de desarrollo de analizadores léxicos al</p>	

generar código optimizado para el reconocimiento de patrones, acelerando así la construcción de software relacionado con el procesamiento de texto estructurado.

4	MATERIALES, EQUIPOS Y RECURSOS EN GENERAL A UTILIZAR
	<ul style="list-style-type: none">• Equipos de cómputo (Nuestros equipos son de gama media)<ul style="list-style-type: none">◦ 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• WLS, Máquina virtual con Ubuntu o S.O Ubuntu 23.04 (En nuestro caso usamos un WLS; Ubuntu en Windows)• Flex o Flex-Old de Ubuntu

5	DESARROLLO
	<p>Primero, verificamos si tenemos instalado flex</p> <div style="background-color: black; color: green; padding: 10px;"><pre>④ freddy024@DESKTOP-GH2FFV4:~/flex\$ flex -v Command 'flex' not found, but can be installed with: sudo apt install flex # version 2.6.4-8build2, or sudo apt install flex-old # version 2.5.4a-10.1 ④ freddy024@DESKTOP-GH2FFV4:~/flex\$ █</pre></div> <p>En el caso de no tenerlo instalado, entonces debemos usar el comando sudo apt install flex</p> <div style="background-color: black; color: green; padding: 10px;"><pre>④ freddy024@DESKTOP-GH2FFV4:~/flex\$ sudo apt install flex [sudo] password for freddy024: Reading package lists... Done Building dependency tree... Done Reading state information... Done The following additional packages will be installed: libfl-dev libfl2 m4</pre></div> <p>Ahora generamos el código para el analizador léxico: La extensión del archivo del analizador léxico, lexer o scanner, es: .l (punto “ele”)</p>

Pasos para crear un analizado léxico con Flex

1. Crear un archivo .l
2. Escribir las definiciones y reglas o patrones que deben seguir los tokens
3. Compilar el archivo .l
4. Se puede probar el scanner dándole de “comer” un archivo.txt con los tokens o caracteres que queremos comprobar

Estructura del código del analizador léxico

Sintaxis para las definiciones

```
%{
    // Definiciones
%}
```

Sintaxis para la sección de reglas del scanner

```
%%
Patrón/Exp. Regular      Instrucción a ejecutar
%%
```

Pattern	It can match with
[0-9]	all the digits between 0 and 9
[0+9]	either 0, + or 9
[0, 9]	either 0, ‘,’ or 9
[0 9]	either 0, ‘.’ or 9
[-09]	either -, 0 or 9
[-0-9]	either – or all digit between 0 and 9
[0-9]+	one or more digit between 0 and 9

[^a]	all the other characters except a
[^A-Z]	all the other characters except the upper case letters
a{2, 4}	either aa, aaa or aaaa
a{2, }	two or more occurrences of a
a{4}	exactly 4 a's i.e, aaaa
.	any character except newline
a*	0 or more occurrences of a
a+	1 or more occurrences of a
[a-z]	all lower case letters
[a-zA-Z]	any alphabetic letter
w(x y)z	wxz or wyz

Esta tabla presenta alguno de los patrones que podemos usar para representar expresiones regulares en nuestro scanner. La presente tabla es recogida de la página: <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>

Del artículo: **Flex (Fast Lexical Analyzer Generator)** escrito por el usuario:
shivani.mittal

Sintaxis para la sección de código del usuario

En esta sección se pueden realizar funciones que sean independientes del analizador léxico.

```
%{
// Definiciones
%}

%%
Reglas/Patrones
%%
```

Ejecución del programa

Podemos ejecutar el programa de la siguiente manera:

1. Guardamos nuestro scanner con extensión .l
2. Utilizamos el comando **gcc lex.yy.v**
3. Ejecutamos el archivo objeto generado **./a.out**
4. Ingresamos el texto a comprobar (puede ser una cadena, un archivo de texto, etc.)

Creemos nuestro primer ejercicio

```
/*
LA-II GRUPO A
EQUIPO 1
ARROYO GOMEZ JOSE ALFREDO
GARCIA HERNANDEZ CESAR
GASCA PALACIO JESUS FERNANDO
GONZALEZ MANCERA CHRISTIAN MANUEL
*/
%{ // Sección de declaraciones
#include <stdio.h> // Cargamos la librería standar de C
%}

DIGIT [0-9]
LETTER [a-zA-Z]
CHAR_SP _|\$ 
POINT \.
INT_TYPE integer
FLOAT_TYPE float
AT @
EMAIL {LETTER}+{AT}{LETTER}+{POINT}{LETTER}+
STATE GT|DF|GD
CURP [A-Z]{4}{DIGIT}{6}(M|H){STATE}[A-Z]{3}({DIGIT}|{LETTER}){2}
IDENTIFIER ({CHAR_SP}|{LETTER})+({CHAR_SP}|{LETTER}|{DIGIT})*
%%

{DIGIT}+    { printf("Integer number: %s\n", yytext); }
{DIGIT}+{POINT}{DIGIT}+    { printf("Float number: %s\n", yytext); }
{INT_TYPE}  { printf("Integer data type: %s\n", yytext); }
{FLOAT_TYPE} { printf("Float data type: %s\n", yytext); }
{EMAIL} { printf("Email Detected: %s\n", yytext); }
```

```

{CURP}    { printf("Curp Detected: %s\n", yytext); }
[\\t\\n] ;// Ignorer white spaces and new lines
.    { printf("Token not recognized: %s\n", yytext); }
{IDENTIFIER} { printf("Identifier Detected: %s\n", yytext); }
%%

int main() {
    yylex();
    return 0;
}

```

```

scanner.l x entry.txt
lexer01 > scanner.l
1  /*
2  LA-II·GRUPO·A
3  EQUIPO 1
4  ARROYO·GOMEZ·JOSE·ALFREDO
5  GARCIA·HERNANDEZ·CESAR
6  GASCA·PALACIO·JESUS·FERNANDO
7  GONZALEZ·MANCERA·CHRISTIAN·MANUEL
8  */
9  %{ // Seccion de declaraciones
10 #include <stdio.h> // Cargamos la libreria standar de C
11 %}
12
13 DIGIT [0-9]
14 LETTER [a-zA-Z]
15 CHAR_SP _|\$|
16 POINT \.
17 INT_TYPE integer
18 FLOAT_TYPE float
19 AT @
20 EMAIL {LETTER}+{AT}{LETTER}+{POINT}{LETTER}+

```

Definiciones léxicas:

DIGIT representa un dígito (0-9).

LETTER representa una letra (a-z o A-Z).

CHAR_SP representa caracteres especiales, como _ (guion bajo) y \$ (dólar).

POINT representa un punto (.)

INT_TYPE representa la palabra clave integer.

FLOAT_TYPE representa la palabra clave float.

EMAIL representa una dirección de correo electrónico simple.

STATE representa los estados **GT**, **DF** y **GD**.

CURP representa una estructura de CURP (Clave Única de Registro de Población).

IDENTIFIER representa la estructura de un identificador.

Reglas léxicas:

{DIGIT}+ coincide con secuencias de uno o más dígitos y las imprime como "Integer number". Es decir, un entero.

{DIGIT}+{POINT}{DIGIT}+ coincide con números flotantes y los imprime como "Float number". Es decir, un numero de punto flotante.

{INT_TYPE} coincide con la palabra reservada integer.

{FLOAT_TYPE} coincide con la palabra reservada float.

{EMAIL} coincide con direcciones de correo electrónico.

{CURP} coincide con CURPS.

[\t\n] se utiliza para ignorar espacios en blanco, tabulaciones y saltos de línea.

. coincide con cualquier otro carácter no reconocido y lo imprime como "Token not recognized".

{IDENTIFIER} coincide con un identificador simple.

Archivo de entrada

```
scanner.l entry.txt
lexer01 > entry.txt
1 chino@gmail.com
2 chino@.com
3 AOGA070802HGTRMLA3
4 AOG9070802HGTRMLA3
5 2014
6 int
7 float
8 double
9 @
10 ?
11 person123
12 123myvar
13 variable
```

En el siguiente archivo podemos observar que reconocerá algunos números, identificadores, CURPS, un correo, etc. Pero en el caso de la segunda CURP, está mal formulada de tal forma que el analizador léxico no la reconocerá. Así mismo para otros casos. Compilemos nuestro programa para ver que genera en la salida.

1. flex scanner.l
2. gcc -o lexer lex.yy.c -lfl
 - a. Genera los archivos: **lexer**, **lex.yy.c**
3. ./lexer < entry.txt
 - a. Ejecutamos el programa y le damos por entrada nuestro archivo **entry.txt**

```
● freddy024@DESKTOP-GH2FFV4:~/flex/lexer01$ flex scanner.l
● freddy024@DESKTOP-GH2FFV4:~/flex/lexer01$ gcc -o lexer lex.yy.c -lfl
● freddy024@DESKTOP-GH2FFV4:~/flex/lexer01$ ./lexer < entry.txt
Email Detected: chino@gmail.com
Identifier Detected: chino
Token not recognized: @
Token not recognized: .
Identifier Detected: com
Curl Detected: AOGA070802HGTRMLA3
Identifier Detected: AOG9070802HGTRMLA3
Integer number: 2014
Identifier Detected: int
Float data type: float
Identifier Detected: double
Token not recognized: @
Token not recognized: ?
Identifier Detected: person123
Integer number: 123
Identifier Detected: myvar
Identifier Detected: variable
○ freddy024@DESKTOP-GH2FFV4:~/flex/lexer01$
```

En la salida podemos observar que:

- Se reconoce correctamente un correo
- Se reconoce un identificador
- No reconoce el carácter @ ni . (punto) debido a que el correo está mal formulado
- Por lo anterior, reconoce el “com” como identificador
- Reconoce un correo correctamente: AOGA070802HGTRMLA3
- El siguiente correo: AOG9070802HGTRMLA3 no lo reconoce como correo debido al error en el 4to carácter el cual debe ser una letra mayúscula.
- Detecta un entero
- Detecta la palabra reservada int y float
- La cadena “double” la detecta como un identificador porque no le indicamos al analizador léxico que es una palabra clave
- No reconoce otros símbolos como: @ o ?
- Identifica identificadores como: person123
- Identifica números si son escritos antes de caracteres con los que debería empezar un identificador: _ | \\$ | [A-Z-a-z]

6

BITÁCORA DE INCIDENCIAS

Fecha	Hora	Descripción de la incidencia	Solución
Martes 26 de septiembre de 2023	17:00	Al compilar, nos marcaba errores en la definición de las reglas. Fue debido a algunos errores en el uso de los patrones o expresiones regulares.	Se hizo un análisis visual para identificar los errores que cometimos al escribir nuestros patrones.

7

OBSERVACIONES

En general creemos que podemos mejorar en el aspecto de definir las expresiones regulares e intentar práctica más las mismas para poder evitar errores que puedan crear conflictos a la hora de definir nuestras palabras reservadas y/o tokens.

8

CONCLUSIONES

Esta práctica nos ayuda a comprender de una mejor manera, el cómo podemos utilizar generadores de analizadores léxicos para comprender el funcionamiento de, precisamente, analizadores léxicos. Así, poco a poco, iremos adquiriendo el conocimiento necesario para poder implementar un analizador léxico en un lenguaje de alto nivel o quizá, de nivel medio como C/C++.

En general estamos satisfechos con esta práctica ya que nos permitió aclarar algunos aspectos que no comprendíamos tan bien de FLEX y YAAC. Ya que una duda es de si eran lo mismo, o hasta que puedo uno es independiente del otro. Si existe una relación entre estos 2 precisamente porque son el análisis léxico y el sintáctico, pero los procesos son independientes y la salida de uno es la entrada del otro.

9

REFERENCIAS BIBLIOGRÁFIA

- Shivani Mittal. (10 de abril del 2023, última actualización). Geeks For Geeks. Flex (Fast Lexical Analyzer Generator). Recuperado el 26 de septiembre del 2023 de <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>

ANEXOS

Agregamos los programas realizados en esta práctica por un link en Google drive:

Analizador léxico simple:

https://drive.google.com/drive/folders/1voXsZUTyLog7okF45wQ4HIsiFdBJBnXd?usp=drive_link

CONCLUSIÓN

En el contexto de la materia de Lenguajes y Autómatas II resulta de vital importancia el papel que juega el análisis léxico y el generador de análisis léxico, ya que es esencial para la comprensión y manipulación de lenguajes y estructuras gramaticales.

Estas herramientas desempeñan una función crítica en el proceso de creación de compiladores, intérpretes y lenguajes de programación al brindar una alta eficiencia y precisión.

Se requiere comprender la teoría sobre el proceso de análisis léxico en el proceso de compilación (por ejemplo de que forma se lleva a cabo) para poder implementarlo de forma práctica.

Entonces, el analizador léxico es el responsable de deshacer el código fuente en una secuencia de tokens (y elimina algunas cosas como espacios en blanco o comentarios), lo que facilita la detección de errores en el código. Por su parte, los generadores de análisis léxico dan paso a automatizar la implementación de analizadores léxicos personalizados, para de esta manera ahorrar recursos de tiempo, esfuerzo e inclusive económicos.

Finalizando, son herramientas esenciales para el desarrollo de compiladores (que a final de cuentas será el proyecto a desarrollar), gracias al desarrollo de esta actividad se tienen sólidos conocimientos sobre el tema.

Referencias Bibliográficas

- Arquitectura de Sistemas UC3M. (s.f.). Tabla Hash. Obtenido de Arquitectura de Sistemas UC3M:
https://www.it.uc3m.es/pbasanta/lasng/course_notes/ch07.html
- Universidad Europea de Madrid. (s.f.). ANÁLISIS LÉXICO. CONCEPTOS BÁSICOS DEL ANÁLISIS LÉXICO. Obtenido de cartagena99:
<https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2-M4-U2-T1.pdf>
- Universitat Jaume I. (2009), A Ingeniería Informática 1126 Procesadores de lenguaje Analizador Léxico. Obtenido de Repositori Universitat Jaume I:
<https://repositori.uji.es/xmlui/bitstream/handle/11023/15877/lexico.apun.pdf>
- Viloria, A (s.f.). Una herramienta para el Análisis Léxico. Lex. Universidad de Valladolid. Recuperado 21 de septiembre de 2023, de <https://www.infor.uva.es/nmluis/talf/docs/aula/A3-A6.pdf>
- Universidad de Málaga. (s.f.). LEX: El analizador léxico. Recuperado 21 de septiembre de 2023, de http://www.lcc.uma.es/ingalvez/ftp/tci/Tutorial_Lex.pdf
- Baverá, F., & Nordio, D. (s.f.). Un Generador de Análisis Léxico Traductores. Recuperado 22 de septiembre de 2023, de <https://dc.exa.unrc.edu.ar/staff/fbaveral/papers/TesisJTLex-Baverá-Nordio-02.pdf>

TECNOLOGICO NACIONAL
DE MEXICO EN CELAYA

A signatura: Lenguajes y autómatas II

Docente: Ricardo González González

2023-09-27 LAII-A

Actividad: Monografía

Análisis Sintáctico

Equipo #1

Integrantes

- Arroyo Gómez José Alfredo (20030029)
- García Hernández Cesar (20030853)
- Gasca Palacio Jesús Fernando (20030606)
- González Mancera Christian Manuel (20030115)

César García Hernández

TABLA DE CONTENIDO

PAG

● Introducción - - - - -	- - - - -	1
● Desarrollo - - - - -	- - - - -	2
► Gramáticas Libres de Contexto y Árboles de Derivación	- - - - -	2
• Definición de gramática libre de contexto - - - - -	- - - - -	2
• Reglas de producción y símbolos terminales y no terminales - - - - -	- - - - -	3
• Representación de gramáticas libres de contexto - - - - -	- - - - -	5
■ Notación de Backus-Naur (BNF) - - - - -	- - - - -	5
■ Derivaciones - - - - -	- - - - -	6
• Árboles de derivación - - - - -	- - - - -	9
■ Construcción de un árbol de derivación a partir de una gramática - - - - -	- - - - -	9
• Mínimo árbol de derivación - - - - -	- - - - -	10
■ Definición de árboles de derivación mínimos - - - - -	- - - - -	10
■ Importancia de los árboles de derivación mínimos en el análisis sintáctico - - - - -	- - - - -	11
• Impacto en la programación actual - - - - -	- - - - -	14
■ Análisis sintáctico y compilación de programas - - - - -	- - - - -	12
► Diagramas de Sintaxis - - - - -	- - - - -	13
• Características - - - - -	- - - - -	13
■ Elementos Básicos - - - - -	- - - - -	14
■ Estructura If - - - - -	- - - - -	15
■ Ejemplo sumatoria - - - - -	- - - - -	15
■ Expresiones Aritméticas - - - - -	- - - - -	16
■ Sintaxis en la Programación - - - - -	- - - - -	16
■ Notación BNF - - - - -	- - - - -	18

○ Práctico	• Diagramas de sintaxis - documentación IBM 7.3	- - - - -	19
	• ¿Qué son las gramáticas libres de contexto?	- - - - -	21
	• ¿Qué es un contexto?		22
	• ¿Sirve el contexto a los propósitos de una gramática que define un lenguaje formal?	- - -	23
○ Ejemplos Árboles de Derivación	- - - - -	- - - - -	24
○ Conclusión	- - - - -	- - - - -	26
○ Referencias Bibliográficas	- - - - -	- - - - -	27

 Cesar Garcia Hernandez

INDICE DE CUADROS Y FIGURAS

• Ejemplo. Árbol Simple	9
• Diagramas Sintaxis. Elementos Básicos	14
• Diagramas Sintaxis. Estructura If	15
• Diagramas Sintaxis. Ejemplo Sumatorio	15
• Diagramas Sintaxis. Expresiones Aritméticas	16
• Diagramas Sintaxis. Sintaxis en la Programación	18
• Diagramas Sintaxis. Notación BNF	19
• Diagramas de sintaxis - documentación IBM 7.3	22
• Figuras. ¿Qué es un contexto?	
• ¿Sirve el contexto a los propósitos de una gramática que define un lenguaje formal?	23
• Diagrama	
• Árbol de derivación sintáctico (Parse Tree)	24
• Árbol de expresión	24
• Árbol de dependencias	25
• Árbol generador de oraciones	25

César García Hernández

ANÁLISIS SINTÁCTICO

INTRODUCCIÓN

El análisis sintáctico, es de vital importancia en todo lenguaje de programación; es un proceso fundamental que actúa como la "puerta de entrada" al entendimiento y la validación de la estructura de un lenguaje. En esta fase crucial, se desentrañan las reglas que rigen la organización y combinación de símbolos en un texto o código, permitiendo determinar si dicho texto se ajusta a las normas gramaticales del lenguaje en cuestión.

A medida que se explora el análisis sintáctico en profundidad, se descubre como las gramáticas formales definen las pautas que deben seguirse para construir expresiones y lenguajes coherentes.

La sintaxis gobierna la disposición adecuada de palabras clave, operadores y operandos, y como los árboles de derivación proporcionan una representación visual de la jerarquía subyacente en la estructura del lenguaje.

Esta fase, a menudo automatizada mediante parsers y herramientas de análisis, es un pilar fundamental en el desarrollo de la programación, garantizando reglas gramaticales establecidas y puedan interpretarse.

DESARROLLO

GRAMÁTICAS LIBRES DE CONTEXTO Y ÁRBOLES DE DERIVACIÓN

Definición de gramática libre de contexto

Una gramática libre de contexto, gramática tipo 2 o también conocida como GIC (Gramática Independiente del Contexto), es una estructura lógica matemática en la forma BNF (Backus Naur); consiste en un conjunto finito de reglas gramaticales que se usan para describir el lenguaje expresado por nuestra gramática en un lenguaje de programación.

Tiene la característica de tener dos tipos de símbolos: los terminales y los no terminales que describiremos más adelante.

Como su nombre lo indica este tipo de gramáticos genera lenguaje libres de contexto, esto quiere decir que genera lenguajes que pueden ser reconocidos por un autómata determinista o no determinista. Recordando lo visto en clases y exámenes anteriores, sabemos que un AFD (Autómata Finito Determinista) tiene la característica de hacer transiciones de un estado, pero puede tener como destino un solo estado, por otro lado un AFND (Autómata Finito No Determinista) puede tener múltiples destinos. Así pues un lenguaje puede ser aceptado por un AFD y/o un AFND si el estado inicial finaliza en un estado de aceptación.

Es decir la gramática que acepta un analizador sintáctico es una gramática libre de contexto; esto es de utilidad debido a que es más fácil reconocer las sentencias aceptadas, en comparación a otras más complejas.

Reglas de producción y símbolos terminales y no terminales

Una gramática libre de contexto está formada por:

- Un alfabeto de símbolos terminales.
- Un conjunto de variables no terminales.
- Un conjunto de reglas de producción.
- Una variable inicial.

Siguiendo la formalización clásica de las gramáticas de Chomsky, una gramática libre de contexto, como en toda gramática formal está definida por reglas y tuplas de producción; estas tuplas están compuestas de N elementos, siendo:

$$G = (V_T, V_N, P, S)$$

Donde tenemos que:

- V_T es un conjunto finito de terminales.
- V_N es un conjunto finito de no terminales.
- P es un conjunto de producciones.
- S es el símbolo inicial y pertenece al conjunto de no terminales.

Cuando hablamos de lenguajes de programación, hablamos de símbolos o caracteres los cuales son utilizados en

las reglas de producción de una gramática formal. Entre estos símbolos encontramos la clasificación en: terminales y no terminales.

• **Símbolos terminales:** Son caracteres o símbolos del alfabeto que pueden aparecer en las reglas de producción, es decir, en la entrada o salida también, no se pueden subducir es decir se respeta al 100% su estructura.

Es decir no cambian mediante las reglas de la gramática, a fin de que el lenguaje formal definido sea el conjunto de caracteres terminales producidos por la gramática, son elementales en la definición de nuestro lenguaje.

• **Símbolos no terminales:** Los símbolos no terminales son aquellos que se pueden reemplazar, es decir hoy cadenas con símbolos terminales y no terminales.

Generalmente los denominados variables sintácticas o simplemente variables. Nuestra gramática incluye un símbolo inicial, un carácter designado del conjunto de no terminales del cual se pueden derivar reglas de producción.

• **Producción:** Una gramática se define mediante reglas de producción las cuales especifican que lexemas reemplazan a otros, es decir que caracteres generan nuestras cadenas.

Los reglas de producción tiene dos miembros, un miembro izquierdo, llamado en ocasiones como "cabeza" el cual representa la cadena a reemplazar, y también un miembro derecho o "cuerpo" que representa la cadena que reemplaza.

Por lo regular las reglas se escriben de la siguiente manera:

$$A \rightarrow w$$

Donde especificamos que A reemplaza a w. En la gramática libre de contexto, cada producción tiene la forma de producción anterior; donde el lado izquierdo de la producción puede aparecer símbolos no terminales, mientras que en el lado derecho de la producción pueden aparecer desde símbolos no terminales, así como terminales. Con la característica de que el símbolo de la derecha debe tener una longitud igual o mayor a l.

Representación de gramáticas libres de contexto

a. Notación de Backus-Naur (BNF)

La notación de Backus-Naur o BNF, es un metalingüaje usado específicamente en gramáticas libres de contexto, ya que es una manera formal de describir lenguajes formales.

Una especificación de BNF es un sistema de reglas de derivación. Para poder usar de manera específica la mayoría de los lenguajes de programación y poder comprender las producciones y reglas, se utilizan algunos metalingüajes, mediante metasímbolos de los cuales se requiere saber su significado.

- :: = Se define como

- | O, también conocido como "o"

- {} Repetición

- [] Opcional

- En esta notación se siguen ciertas convenciones entre las que están:
- Los no terminales se escriben entre paréntesis angulares <>.
 - Los terminales se representan con cadenas de caracteres sin paréntesis angulares.
 - El lado izquierdo de cada regla debe tener solamente un no terminal.
 - El símbolo "se define como" ::=, se utiliza en varias producciones donde se pueden dar varias producciones de un solo lado izquierdo, es decir, $\langle A \rangle ::= \langle b \rangle$; $\langle A \rangle ::= \langle c \rangle$, etc. Donde podemos escribir que $\langle A \rangle ::= \langle b \rangle | \langle c \rangle | \dots$ siendo A, b y c no terminales.

Ejemplos

- Definición de un dígito:

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

- Definición de un número entero sin signo:

$$\langle \text{numero_sin_signo} \rangle ::= \langle \text{digito} \rangle | \langle \text{numero_sin_signo} \rangle \langle \text{digito} \rangle$$

- Una dirección postal contiene un nombre, seguido de la dirección, seguido del apartado postal:

$$\langle \text{direccion_postal} \rangle ::= \langle \text{nombra} \rangle \langle \text{direccion} \rangle \langle \text{apartado_postal} \rangle$$

b. Derivaciones

Una regla de producción puede considerarse como equivalente a una regla de reescritura, es decir tenemos un cambio o sustitución donde el no terminal de la izquierda es sustituido por la cadena terminal o no terminal del lado derecho en cada producción.

Como se mencionó anteriormente el lado derecho puede ser cualquier pseudocadena. Derivada de una pseudocadena B, como vimos arriba \Rightarrow

Dentro del lado izquierdo, puede haber varios no terminales, que pueden ser reescritos, esto da lugar a varias derivaciones posibles.

Existen dos formas para poder describir como desde una gramática, una cadena puede ser derivada desde el símbolo inicial. Esto se le conoce como inferencia recursiva, y se da usando las producciones. La derivación y la inferencia son equivalentes, por lo que podemos decir que se puede inferir si una cadena de los terminales está en el lenguaje

- **Derivación por la izquierda:** Este tipo de derivación se da cuando la reescritura se realiza sobre el no terminal más a la izquierda de la cadena inicial.
- **Derivación por la derecha:** Este tipo de derivación se da cuando la reescritura se realiza sobre el no terminal más a la derecha de la cadena inicial.

La diferencia entre estos dos tipos de derivación es la aplicación que tiene los analizadores sintácticos, ya que la transformación del carácter de entrada es definida por las reglas para cada producción, de modo que es importante saber qué derivación aplica el analizador, pues nos determina el orden en el que nuestro código es ejecutado.

Cualquier derivación tiene una derivación a la izquierda y a su vez una a la derecha, es decir se puede llegar a un resultado. Si se cumple con las reglas.

Ejemplos Derivación

Usaremos una gramática simple para formar frases en un lenguaje.

Gramática:

- $\langle \text{Frase} \rangle \rightarrow \langle \text{Sujeto} \rangle \langle \text{Verbo} \rangle \langle \text{Objeto} \rangle$
- $\langle \text{Sujeto} \rangle \rightarrow \text{"Juan"} \mid \text{"María"} \mid \text{"Pedro"}$
- $\langle \text{Verbo} \rangle \rightarrow \text{"come"} \mid \text{"bebe"} \mid \text{"lee"}$
- $\langle \text{Objeto} \rangle \rightarrow \text{"pizza"} \mid \text{"ojo"} \mid \text{"libro"}$

Cadena inicial: "Juan come pizza"

* Derivación por la izquierda:

Lr Frase (Inical)

2r $\langle \text{Sujeto} \rangle \langle \text{Verbo} \rangle \langle \text{Objeto} \rangle$; Usando la regla $\langle \text{Frase} \rangle \Rightarrow \langle \text{Sujeto} \rangle \langle \text{Verbo} \rangle \langle \text{Objeto} \rangle$

3r "Juan" ; Usando la regla $\langle \text{Sujeto} \rangle \Rightarrow \text{"Juan"}$

4r "Juan" "come" ; Usando la regla $\langle \text{Verbo} \rangle \Rightarrow \text{"come"}$

5r "Juan" "come" "pizza" ; Usando la regla $\langle \text{Objeto} \rangle \Rightarrow \text{"pizza"}$

* Derivación por la derecha:

Lr Frase (Inical)

2r $\langle \text{Sujeto} \rangle \langle \text{Verbo} \rangle \langle \text{Objeto} \rangle$; Usando la regla $\langle \text{Frase} \rangle \Rightarrow \langle \text{Sujeto} \rangle \langle \text{Verbo} \rangle \langle \text{Objeto} \rangle$

3r "pizza" ; Usando la regla $\langle \text{Objeto} \rangle \Rightarrow \text{"pizza"}$

4r "come" "pizza" ; Usando la regla $\langle \text{Verbo} \rangle \Rightarrow \text{"come"}$

5r "Juan" "come" "pizza" ; Usando la regla $\langle \text{Sujeto} \rangle \Rightarrow \text{"Juan"}$

Ambas derivaciones generaron la misma frase "Juan come pizza" utilizando la misma gramática. La diferencia radica en el orden en que se aplican las reglas de producción, lo que demuestra la versatilidad de las gramáticas libres de contexto para expresar estructuras de lenguaje en diferentes direcciones.

Arboles de derivación

Los árboles de derivación son una representación gráfica utilizada en la teoría de la gramática y el análisis sintáctico. Estos árboles muestran cómo una oración o frase se deriva o se descompone en sus componentes gramaticales, siguiendo las reglas y la estructura de una gramática dada.

a. Construcción de un árbol de derivación a partir de una gramática

Como se mencionó antes un árbol de derivación sirve para representar una derivación, este se representa mediante un árbol de forma invertida y para su construcción se define el árbol de la siguiente manera:

- La raíz del árbol es el carácter inicial de la gramática.
- Los nodos inferiores o también conocidos como ocultos están etiquetados por los símbolos no terminales.
- Las hojas (nodos sin hijos) están etiquetados por símbolos terminales.
- Si un nodo interior etiquetado posee hijos otros nodos, se dice que es una producción de la gramática.

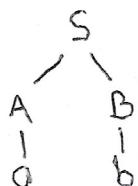
Ejemplo. Árbol simple

Gramática libre de contexto

$$S \rightarrow AB$$

$$A \rightarrow a$$

$$B \rightarrow b$$



En este ejemplo la gramática tiene una única regla de producción $S \rightarrow AB$. Esto significa que una derivación válida comenzará con un símbolo no terminal "S", y este se expande en los símbolos no terminales "A" y "B". Además de tener dos símbolos terminales "a" y "b".

Así pues podemos definir los pasos en:

1: Definir una gramática.

2: Identificar los símbolos y reglas de la gramática.

3: Aplicar las reglas de derivación.

4: Representar el árbol de derivación.

Es importante destacar que los árboles de derivación pueden ser más o menos detallados dependiendo de la gramática y la especificación de las reglas. Cada rama del árbol muestra una posible secuencia de pasos para derivar la oración, lo que permite visualizar la estructura sintáctica de una manera clara y concisa.

Estos árboles son fundamentales en el análisis y la comprensión de la gramática de un lenguaje, así como en la generación automática del texto y otras aplicaciones del procesamiento del lenguaje natural.

Mínimo árbol de derivación

a. Definición de árboles de derivación mínimos

Un mínimo árbol de derivación, también conocido como árbol sintáctico mínimo o árbol de análisis sintáctico, es una representación gráfica de la estructura de una frase o enunciado en el análisis sintáctico de un lenguaje.

Esta representación muestra cómo los distintos componentes de la frase están relacionados entre sí y cómo se derivan a partir de reglas gramáticas.

El árbol de derivación mínimo es importante en el estudio de la lingüística y la gramática, ya que ayuda a comprender y analizar la estructura sintáctica de una sentencia.

Permite identificar la función y la relación de cada palabra o elemento en la oración, así como también facilita la identificación de errores gramaticales o ambiguos en el texto.

El análisis del árbol de derivación mínimo también es utilizado en el procesamiento del lenguaje natural y la programación, ya que permite la construcción de algoritmos y sistemas para entender el significado y la estructura gramatical de un texto.

b. Importancia de los árboles de derivación mínimos en el análisis sintáctico

En resumen, el mínimo árbol de derivación es una representación gráfica que muestra la estructura sintáctica de una oración, siendo importante para el análisis del lenguaje natural y la programación. Ayuda a comprender la relación y función de cada elemento en una oración, y permite detectar errores gramáticos y ambiguos.

Impacto en la programación actual

Los gramáticos libres de contexto son una herramienta fundamental en el diseño de lenguajes de programación, permiten definir la estructura sintáctica de un lenguaje de manera formal y precisa. En el diseño de lenguajes, se utilizan para especificar cómo se deben escribir los programas y como estos deben analizarse y compilarse.

Así como vimos anteriormente los gramáticos libres de contexto constan de reglas que describen la relación entre símbolos terminales (palabras clave y símbolos) y no terminales (variables y expresiones). Esto es la base para la creación de analizadores sintácticos, como los parsers que verifican

Si un programa escrito en un lenguaje complejo con su estructura sintáctica. Esto es esencial para evitar errores de sintaxis y facilitar la detección temprana de problemas en el código fuente.

En resumen, los gramáticos libres de contexto son esenciales en el diseño de lenguajes de programación porque proporcionan una base formal para definir la sintaxis del lenguaje, permiten el análisis y la verificación automática de los lenguajes. Esto facilita la creación de compiladores e intérpretes.

b. Análisis sintáctico y compilación de programas

El análisis sintáctico y la compilación son dos etapas ejecutadas en la ejecución de programas de computadora.

El análisis sintáctico es la primera fase de la traducción de un programa escrito en un lenguaje de alto nivel a un código ejecutable. Su objetivo principal es verificar que el programa cumpla con las reglas de sintaxis definidas en el lenguaje de programación. Esto implica comprobar la estructura y organización correcta de las instrucciones.

Para lograr esto se debe basar en una gramática formal que describe la sintaxis del lenguaje. Si el código tiene errores sintácticos, se generan mensajes de error.

La compilación es la segunda fase y sigue al análisis sintáctico. Consiste en la traducción del código fuente validado en un lenguaje de alto nivel a un código intermedio o código máquina específico de la arquitectura de la computadora de destino. Durante la compilación se realizan optimizaciones para mejorar la eficiencia del programa. El compilador realiza una asignación de memoria y resolución de direcciones.

Diagramas de sintaxis

Los diagramas sintácticos también conocidos como diagramas de sintaxis o incluso como diagramas de ferrocarril son una forma de representar una gramática libre de contexto.

Podemos ver estos diagramas como una alternativa gráfica para la forma BNF.

Los diagramas de sintaxis también son una representación diferente a los árboles de derivación, que comúnmente se conocen por especificar gramáticas de cualquier tipo, algunos sitios web hacen énfasis en el tipo 2, pero pueden especificar cualquiera.

Características

Una de las características de este tipo de esquema es que de esta forma se ven las derivaciones de manera instantánea y gráfica.

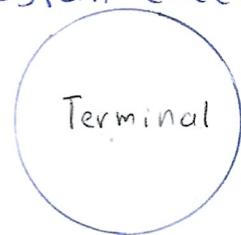
Así que esto facilita de manera visual como se representa la gramática de un lenguaje.

Podemos encontrar diversas representaciones, dependiendo de los autores, de los recursos de donde se obtienen entre otras circunstancias.

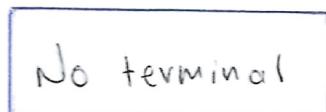
- Símbolo terminal : (palabra o tokens), se reconoce un símbolo terminal cuando tiene una entidad propia que puede describirse y así identificarse por sí mismo.

ELEMENTOS BÁSICOS

Estos están encerrados en un círculo y su nombre dentro de él.

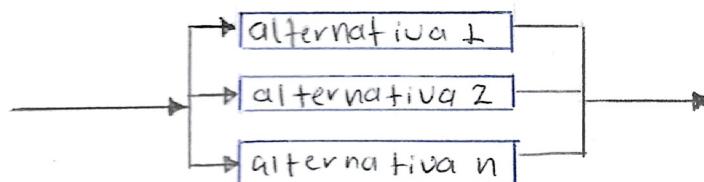


- Símbolo no terminal: se reconoce un símbolo no terminal cuando un símbolo necesita ser definido a través de una regla o producción para ser comprendido. Se representa por su nombre encerrado en un rectángulo.

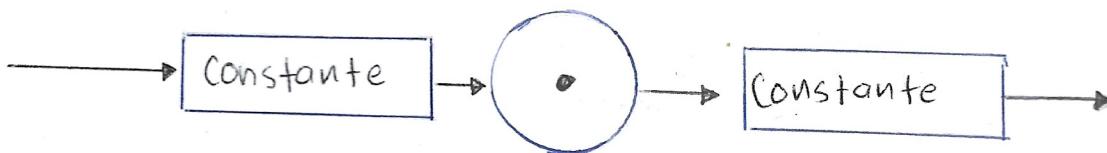


• Producciones

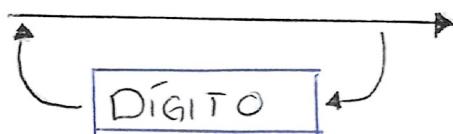
- Producciones con varias alternativas: podemos tener desde 1 hasta n alternativas.



- Producciones concatenadas: los símbolos se deben concatenar uno a continuación del otro, pueden ser símbolos terminales o no terminales.

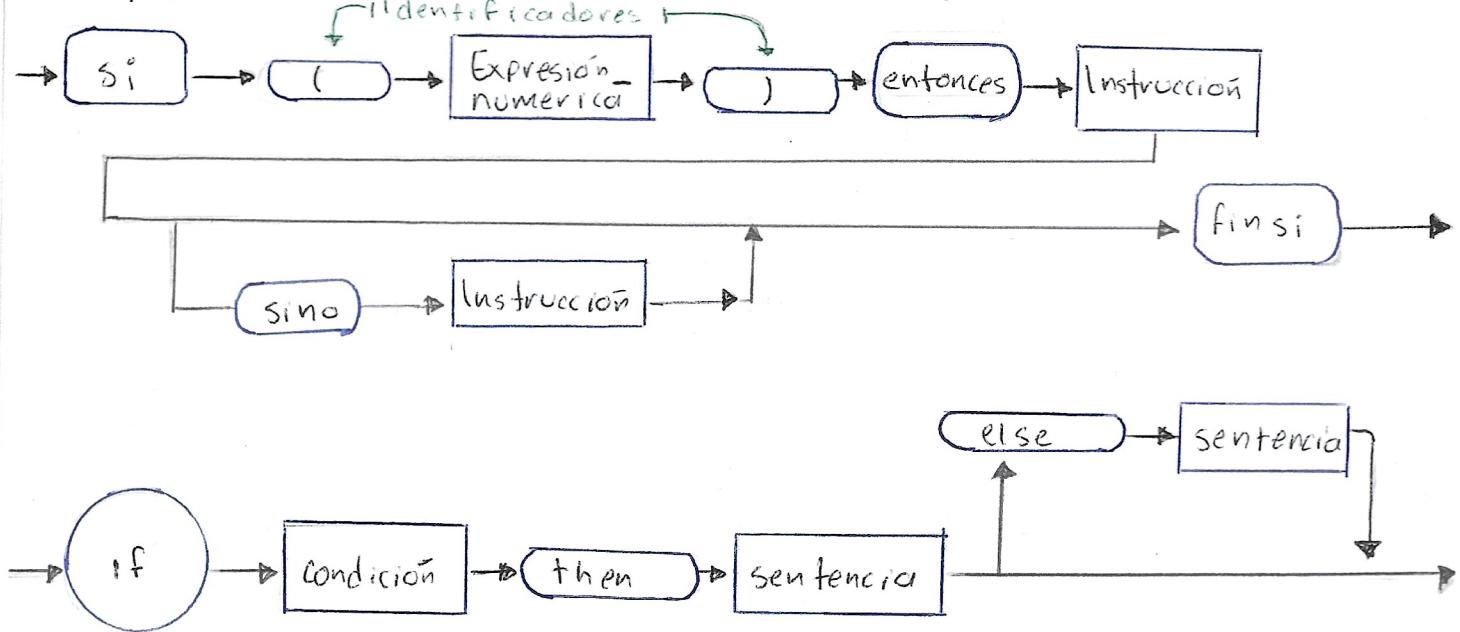


- Producciones con repeticiones: pueden tener 0, una o más repeticiones de un símbolo.



ESTRUCTURA IF

Los siguientes ejemplos podemos entenderlos como una estructura "if" en los lenguajes de programación.

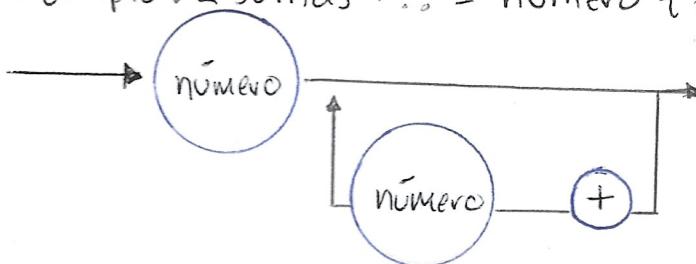


Al usar estos diagramas sintácticos somos capaces de representar la sintaxis de un lenguaje y podemos utilizarlos en lugar del conjunto de producciones en BNF.

Los diagramas sintácticos tienen la ventaja de que son claros y concisos, ya que facilitan la comprensión del lenguaje. Pero esto también tiene un inconveniente que es la dificultad de representación.

EJEMPLO SUMATORIA

Ejemplo: <sumas> :: = número { + número }



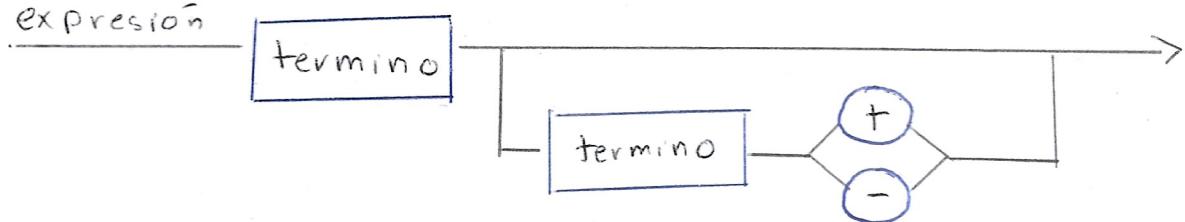
En el siguiente ejemplo podemos ver la misma operación, pero esta representada con diagrama de sintaxis y con notación BNF. Este ejemplo deja en claro la ventaja visual de los diagramas.

Es importante tener en cuenta que todo diagrama posee un punto de entrada (generalmente situado a la izquierda) y un punto de salida (a la derecha).

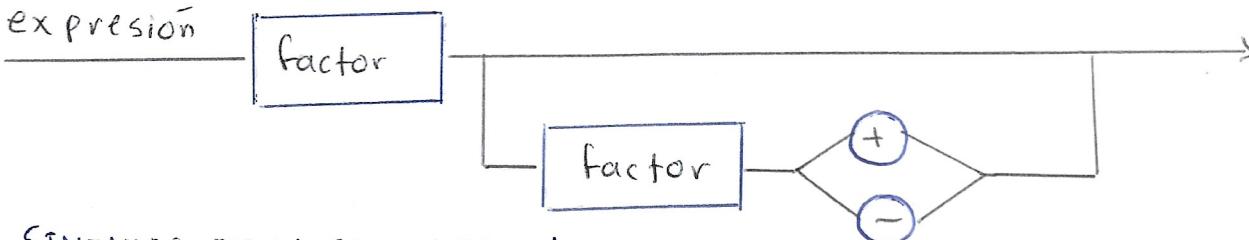
EXPRESIONES ARITMÉTICAS

A continuación, se muestran los siguientes diagramas que muestran cómo se podrían usar para representar expresiones aritméticas.

expresión

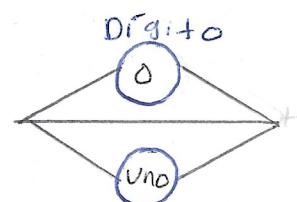
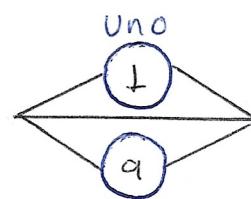
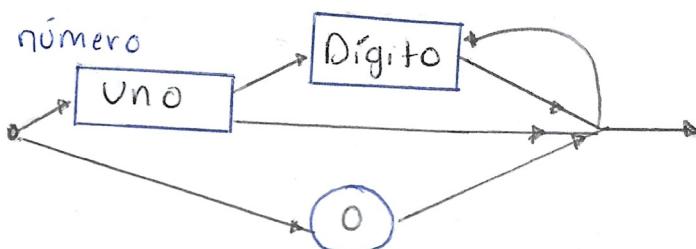


expresión

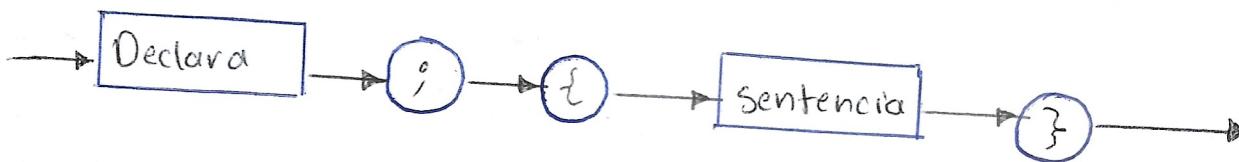


SINTAXIS EN LA PROGRAMACIÓN

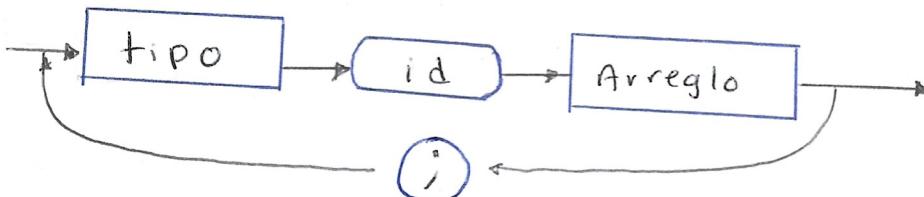
El siguiente ejemplo hace referencia a como se usarían los diagramas de sintaxis en el uso de la programación.

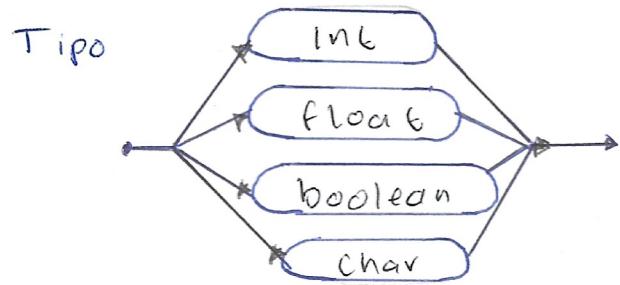
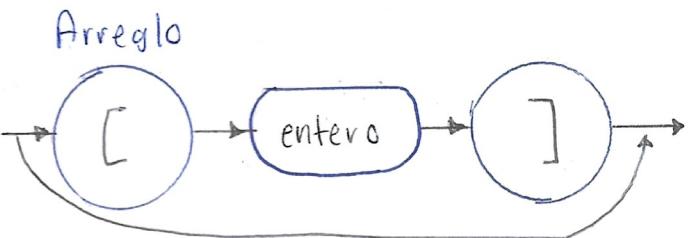


programa

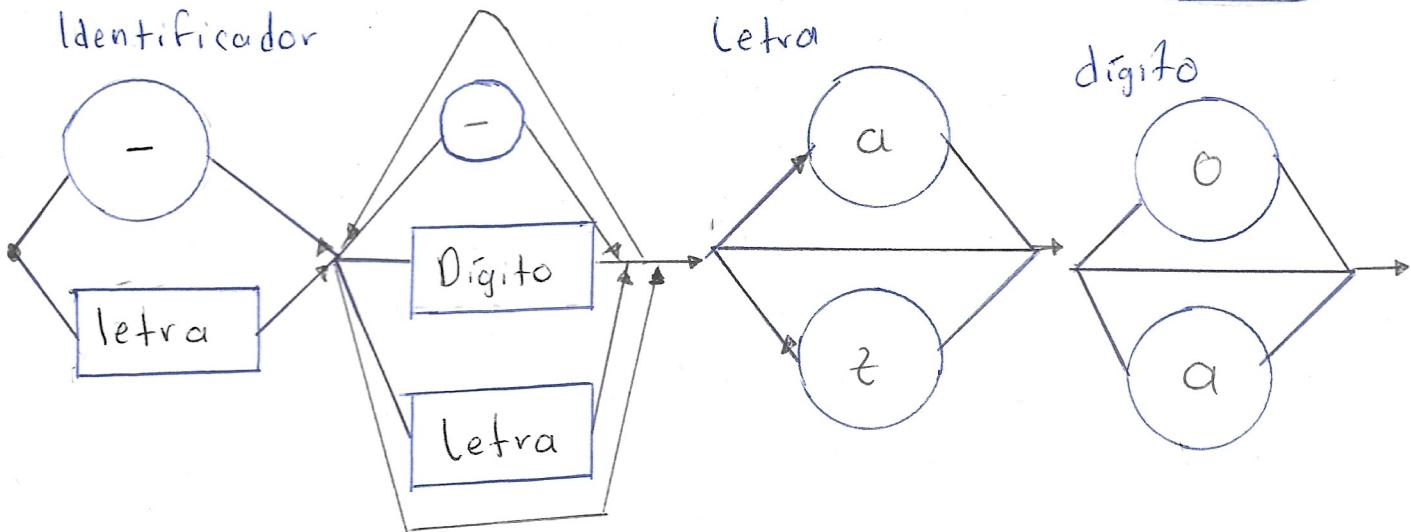


Declaro





Identificador



Letra

dígito

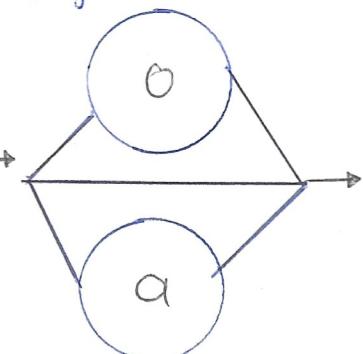


Diagrama de sintaxis para la instrucción de asignación de un valor a una variable

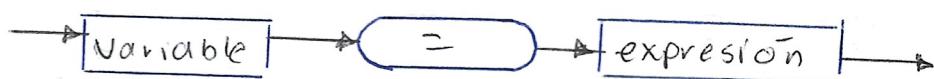
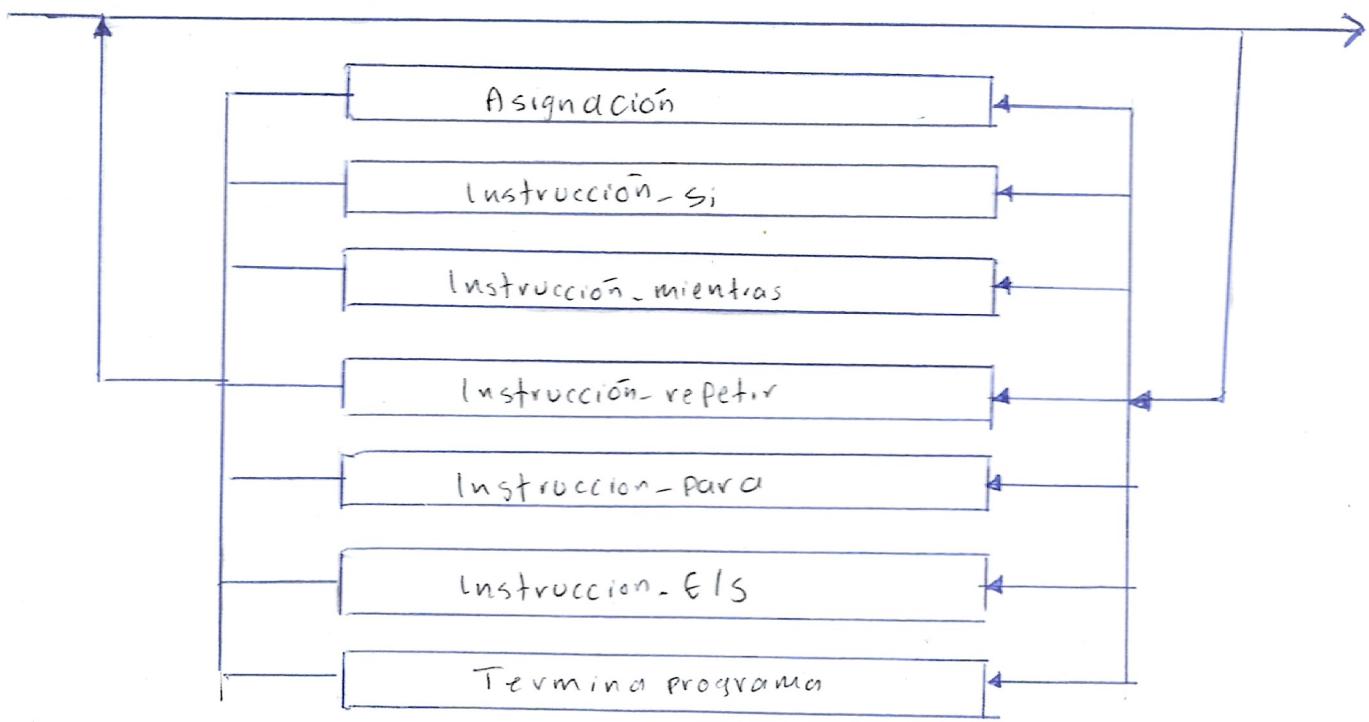
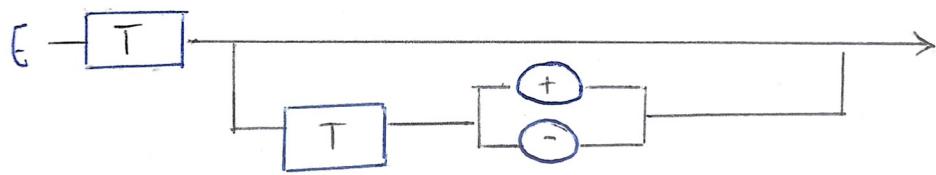


Diagrama de sintaxis para diferentes casos en programación

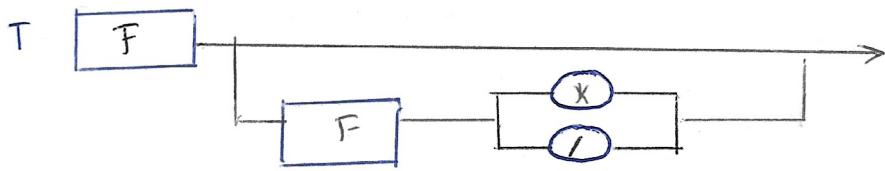


NOTACIÓN BNF

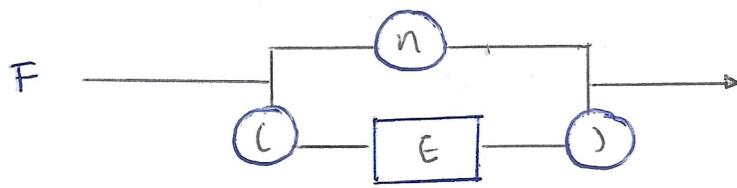
Diagramas de sintaxis con notación BNF



$$\begin{aligned} E ::= & E + T \mid \\ & E - T \mid T \end{aligned}$$



$$\begin{aligned} T ::= & T * F \mid \\ & T / F \mid F \end{aligned}$$



$$F ::= (E) \mid n$$

Diagramas de sintaxis - documentación IBM 7.3

Normas - Leer los diagramas de izquierda a derecha y de arriba abajo, siguiendo el recorrido de la linea.

- »» principio del diagrama.
- ? Indica que la sintaxis continua en la linea siguiente.
- Indica que la sintaxis continua de la linea anterior.
- » Final del diagrama de sintaxis.
- └ Comienzo del diagrama.
- └ Final del diagrama.

• Los elementos obligatorios aparecen en la linea horizontal.

>>-elemento-obligatorio ----- ><

• Los elementos opcionales aparecen debajo de la ruta principal.

>>-elemento-obligatorio -----
+-- elemento-optional - +-----><
+-- elemento-optional - +-----><

en algunos casos lo podemos encontrar encima sólo para efectos de legibilidad.

>>-elemento-obligatorio - +-----><
+-- elemento-optional - +-----><

• Núltiples elementos, aparecen verticalmente en una pila.

- Si uno de los elementos debe elegirse, aparecen en la ruta principal.

>>-elemento-obligatorio - + Opción-obligatoria +-----><
+-- Opción-obligatoria - +-----><

- si los elementos son opcionales, aparecen debajo de la principal.

>>-elemento-obligatorio - +-----><
+-- Opción-optional - +-----><
+-- Opción-optional - +-----><

- Si uno de los elementos es el valor predeterminado, aparecerá encima de la ruta principal y las otras opciones debajo.

>>-elemento-obligatorio - + - Opción-predeterminada +-----><
+-- Opción-optional - +-----><
+-- Opción-optional - +-----><

- Si un elemento opcional tiene un valor predeterminado cuando no se especifica, el valor aparece encima de la ruta principal.

- elemento de repetición, si una flecha vuelve a la izquierda sobre la principal.

> -elemento- obligatorio —— elemento-repetible +-->

César García Hernández

Los elementos anteriores fueron recuperados de la documentación de IBM
IBM documentation, (s.f.). <https://www.ibm.com/docs/es/i/17.3/topic=Programming-how-read-syntax-diagrams>

¿Qué son las gramáticas libres de contexto?

Podemos usar como ejemplo obtener una gramática libre de contexto para el siguiente lenguaje.

$$L = \{ a^m b^n c^p d^q \mid m+n \geq p+q \}$$

Considerar la secuencia de a's seguida de una secuencia de b's seguida de una secuencia de c's seguida de una secuencia de d's

$a^{aa} b^{bb} c^{cc} d^{dd} \checkmark \quad abc d \checkmark \quad aaabb d ccc a \times$

Tanto m y n como p y d podrían ser 0, siempre y cuando se cumpla que: $m+n \geq p+q$

$a^{aa} b^{bb} c^{cc} d^{dd} \checkmark \quad b^{bbb} b^{bb} c^{dd} \checkmark \quad b^{bbb} \checkmark \quad a^{ab} b^{bb} \checkmark$

Usando la cadena vacía podemos exemplificar de la siguiente forma

Tiene cero a's $\rightarrow m=0$ Tiene cero b's $\rightarrow n=0$

Tiene cero c's $\rightarrow p=0$ Tiene cero d's $\rightarrow q=0$

$$\begin{aligned} m+n &\geq p+q \\ 0+0 &\geq 0+0 \end{aligned}$$

Se tiene que cumplir que el número de a's (m) más el número de b's (n) sea al menos igual al número de c's (p) más el número de d's (q).

$\underbrace{a^{aa} b^{bb}}_5 \underbrace{c^{cc} d^{dd}}_5 \checkmark \quad \underbrace{b^{bbb} b^{bb}}_5 \underbrace{c^{dd}}_3 \checkmark \quad \underbrace{a^{ab} b^{bb}}_7 \checkmark$

$$m+n \geq p+q$$

Cada vez que se pone una d se pone al menos una a o b, cada vez que se pone un c, se pone al menos una a o b.

$$S \rightarrow A S d \mid B x d \mid A Y c \mid B z c \mid A \mid B \mid A B \mid \epsilon$$

$$A \rightarrow a A \mid a$$

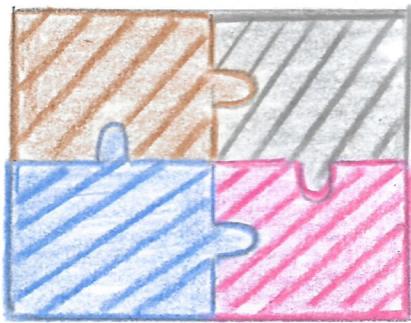
$$B \rightarrow b B \mid b$$

$$X \rightarrow B x d \mid B z c \mid \epsilon$$

$$Y \rightarrow A Y c \mid B t c \mid \epsilon$$

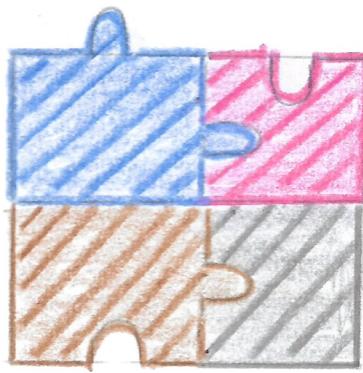
$$Z \rightarrow B z c \mid \epsilon$$

¿Qué es un contexto?

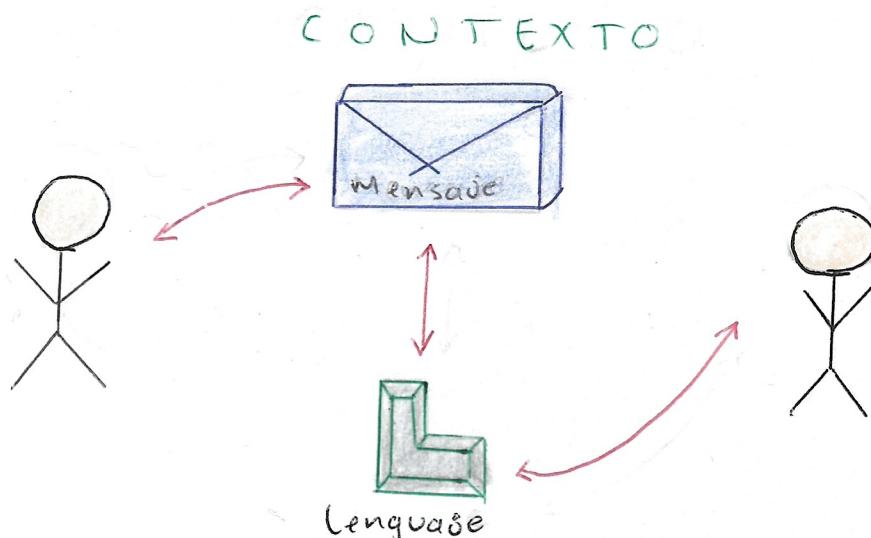


Con contexto

JS



Sin contexto



El contexto es la información que se proporciona para entender el significado y tener más comprensión sobre algo.

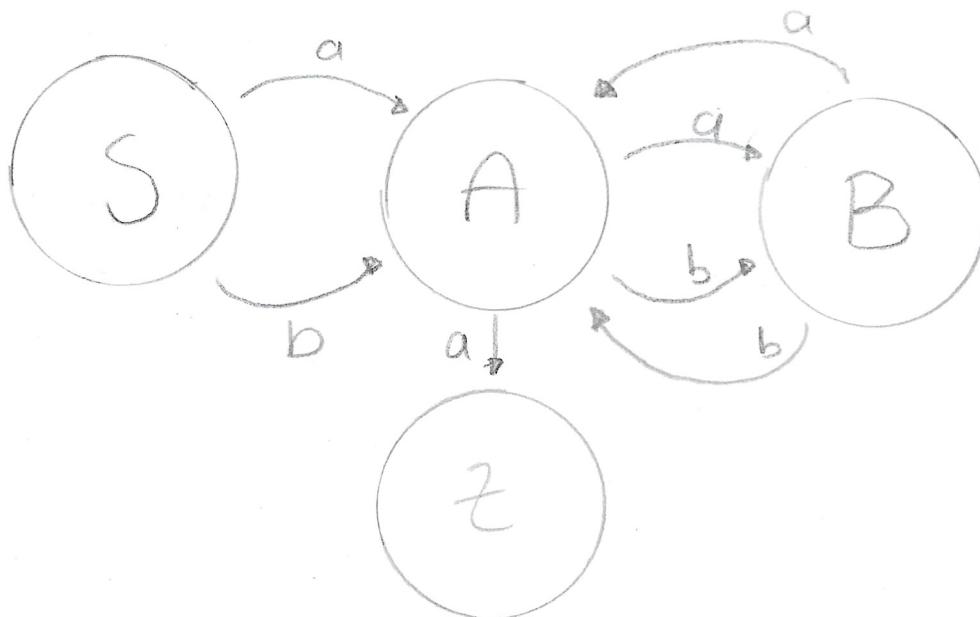
El contexto en los lenguajes de programación se refiere a la forma en que el entorno o las circunstancias influyen en cómo interpretamos los problemas que queremos resolver usando como herramienta de un lenguaje de programación.

○ ¿Sirve el contexto a los propósitos de una gramática que defina un lenguaje formal?

Dentro de un lenguaje formal, la gramática es vista como un conjunto de reglas que permiten la formación de secuencias correctas de símbolos. Estas reglas son estrictas y ya están predefinidas.

El contexto es esencial para los propósitos de una gramática que define un lenguaje formal. Establece las reglas y restricciones que se aplican para formar secuencias correctas de símbolos. Puede influir en cómo se interpretan estas secuencias de símbolos.

- $S \rightarrow aA$
- $S \rightarrow bA$
- $A \rightarrow aB$
- $A \rightarrow bB$
- $A \rightarrow a\zeta$
- $B \rightarrow aA$
- $B \rightarrow bA$



EJEMPLOS ÁRBOLES DE DERIVACIÓN

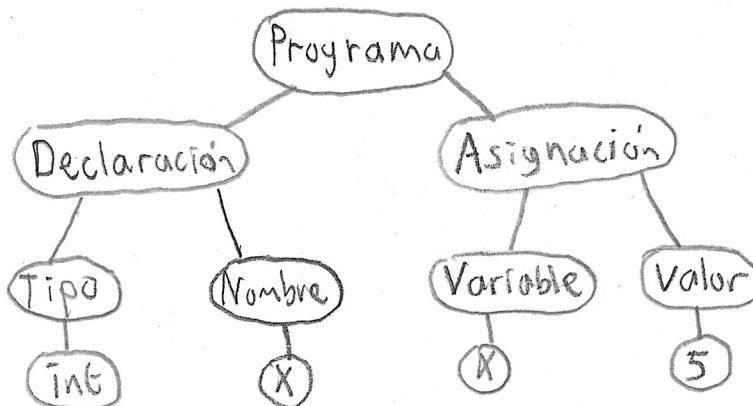
1r Árbol de derivación sintáctico (Parse Tree)

En la compilación de código fuente, se utiliza para representar la estructura sintáctica del programa. Los nodos del árbol representan las construcciones gramaticales, como expresiones, declaraciones y bucles, lo que facilita la verificación y la generación de código intermedio.

o <Programa> → <Declaración> | <Asignación>

o <Declaración> → Tipo : [int | char | double | float], Nombre : [Identificador]

o <Asignación> → Variable : [Identificador], Valor



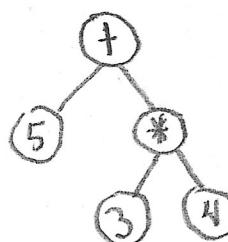
2r Árbol de expresión

En la optimización de código y evaluación de expresiones matemáticas.

o <Expresión> → |+|-|*|/|

o <Número> → |0|1|2|3|4|5|6|7|8|9|

o <Operación> → <Número>|<Expresión>|



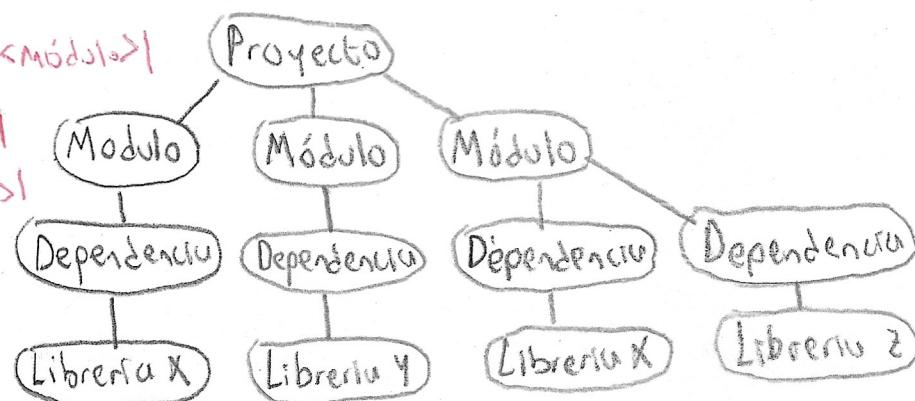
3F Árbol de dependencias

En la gestión de paquetes y dependencias en proyectos de software, se usa para visualizar las relaciones de dependencia entre módulos.

$\alpha \text{Proyecto} \rightarrow \langle \text{Módulo} \rangle | \langle \text{Módulo} \rangle |$

$\langle \text{Módulo} \rangle \rightarrow \langle \text{Dependencia} \rangle | \langle \text{Dependencia} \rangle |$

$\langle \text{Dependencia} \rangle \rightarrow \text{Librería}$



4F Árbol generador de oraciones

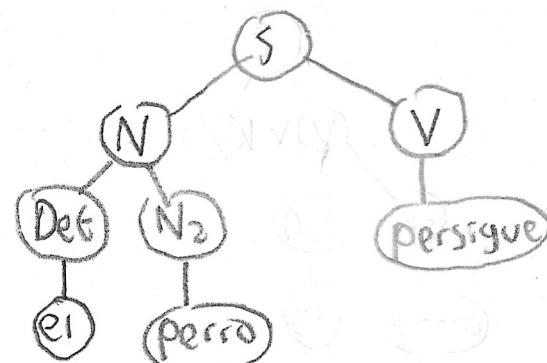
$\langle S \rangle \rightarrow \langle N \rangle < V >$

$\langle N \rangle \rightarrow \langle \text{Det} \rangle \langle N_2 \rangle$

$\langle \text{Det} \rangle \rightarrow \text{el} | \text{un} | \text{mi} | \text{tu} |$

$\langle N_2 \rangle \rightarrow \text{perro} | \text{gato} | \text{humano} | \text{casal}$

$\langle V \rangle \rightarrow \text{persigue} | \text{ataca} | \text{corre} | \text{fiebre}$



CONCLUSIÓN

Como se vio a lo largo de esta monografía; la gramática libre de contexto y los árboles de derivación son conceptos esenciales para el desarrollo de lenguajes de programación y sistemas de análisis sintáctico.

Su aplicabilidad en la programación actual es evidente, ya que permiten definir la estructura de un lenguaje y analizar la validez sintáctica de un programa. A través de la representación de gramáticas y árboles de derivación, establecen un marco formal para el diseño y desarrollo de los lenguajes de programación, sumando herramientas como los diagramas de sintaxis que desempeñan y proporcionan una representación visual de la estructura gramatical de nuestro lenguaje.

Es importante conocer estas estructuras ya que ayudan a los lingüistas, programadores a comprender mejor cómo se organizan las frases y caracteres de un lenguaje, lo que a su vez facilita la identificación de patrones gramaticales, errores y ambigüedades que puedan suceder.

Referencias bibliográficas

- Hopcroft, John E., Jeffrey D. Ullman, y Rajeev Motwani. Introducción a la teoría de autómatas, lenguajes y computación. Pearson Educación, 2008.
- Aho, Alfred V., Monica S. Lam, Ravi Sethi y Jeffrey D. Ullman. Compiladores: principios, técnicas y herramientas. Pearson Educación, 2008.
- Russinovich, Mark E., David A. Solomon y Alex Ionescu. Windows Internals, Parte 2 (Developer Reference). Microsoft Press, 2012.
- Gramáticas libres de contexto y árboles de derivación (<https://www.cs.cornell.edu/courses/cs3110/2018fa/lectures/08-context-free-grammars.pdf>), 2018
- Gramáticas libres de contexto y árboles de derivación (<https://www.cs.cmu.edu/~sujis-112/lectures/06-context-free-grammars.pdf>), 2015
- IBM documentation (s.f) <https://www.ibm.com/docs/en/17.3?topic=programming-how-read-syntax-diagrams>
- Kris (2014, 3 octubre). PPT - Análisis sintáctico Power Point Presentation, Free Download, ID:5152262. Slideserve. <https://www.slideserve.com/kris/analisis-sintactico>
- Lingüística matemática clase 2 (s.f) PPT. <https://es.slideShare.net/jboudabud/linguistica-matematica-clase-2>
- Milo-Scorpio (s.f) Diagramas de sintaxis. <http://lenguajesyautomatas1.unidad6-9.blogspot.com/2016/04/diagramas-de-sintaxis.html>

- Romero, S. (s.f) Diagramas de sintaxis, prezi.com <https://prezi.com/k1t9iuwtizbt/diagramas-de-sintaxis/>
- SSL UTN-FRT - (2020, 3 septiembre) Clase 28 Diagramas sintaxis BNF sin carátula [Video] YouTube. <https://www.youtube.com/watch?v=UVzg10HNxZA>

Notación BNF

La notación BNF o Backus-Naur Form es una notación formal que permite definir la gramática de un lenguaje. Más específicamente lenguajes de programación.

Características:

- Uso de meta-símbolos
- Capacidad de especificar la sintaxis de un lenguaje
- Cuenta con una versión extendida que agrega más meta-símbolos para aumentar la expresividad
- Puedes representar cualquier lenguaje de programación con esta notación

Meta-símbolos BNF:

$::=$ Definición

| OR

{ } Repetición

[] Opcional

* Terminales

↳ 'if', 'while', '15', '32', ...
ejemplos

Meta-símbolos BNF-E:

? Opcional

+ Al menos una vez

* Cero o más veces

Ejemplos

BNF

Gramática para representar cadenas de caracteres con minúsculas.

Cadena ::= minúscula | minúscula cadena

minúscula ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l'
 | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w'
 | 'x' | 'y' | 'z'

Acepta: perro gato rata carrera abc da the zxí

BNF - E

Gramática para representar listas de números enteros con signo:

lista ::= entero Σ ', ' entero Σ

entero ::= [$-$] dígito Σ dígito Σ

dígito ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Acepta:

- 1, 2, 3, 4
- 5
- 10, 110, -453, 3243
- -14

Información recuperada de fing.edu.uy/inco/cursos/teoleng/obligatorio/presentacion.pdf

NOTACIÓN BNF

Las gramáticas son formas de representar la estructura de un lenguaje formalmente, el enfoque de la investigación es desde un punto de vista referente a la representación de lenguajes formales, aunque es importante mencionar que también es posible usar gramáticas para representar lenguajes naturales como el español. El objetivo de la gramática es clarificar y comunicar la estructura, ayudándonos a entender el lenguaje en cuestión, de esta manera, es más fácil implementarlo a un compilador o a un intérprete.

El nombre completo de la notación BNF es Backus-Naur-Form (por John Backus diseñador de FORTRAN y Peter Naur informático danés), esta notación nos ayuda a dar forma y estructura al lenguaje mediante la definición de reglas y la evaluación de cadenas ingresadas a un compilador, si se valida, cumple con las reglas establecidas en el lenguaje cumpliendo con BNF.

Cada lenguaje cuenta con BNF, sin importar su antigüedad pues todos se ven regidos por reglas léxicas, sintácticas y semánticas, de hecho, el compilador busca errores de estos tipos con ayuda de BNF, más específicamente de tipo sintácticos. Brinda herramientas que permiten llevar a cabo una descripción de forma clara y concisa sobre el como se forman los elementos de la estructura sintáctica mediante el uso de reglas y símbolos.

Elementos de BNF

Se apoya de ciertos elementos representados como metasímbolos que cumplen con funciones específicas, explicándose

con más detalle en la siguiente tabla:

Metasímbolo	Significado
$::=$	Define como (igualdad)
	Opción o alternativa
{ }	Elementos incluidos pueden ser repetidos (repetición)
[]	Elementos incluidos pueden o no ser usados (opción)
()	Agrupar elementos

Los símbolos no terminales son encerrados entre "<>", formando reglas de producción se vería de la siguiente manera:

$\langle \text{digito} \rangle ::= 0|1|2|3|4|5|6|7|8|9 \rightarrow$ los números en sí ya son terminales pues no se encuentran entre "<>".

$\langle \text{entero} \rangle ::= \langle \text{digito} \rangle | \langle \text{digito} \rangle \langle \text{entero} \rangle$

Extrapolando el conocimiento a la aplicación de un compilador, podríamos tener ejemplos de mayor complejidad que nos permitan dar reconocimiento a diversas instrucciones dadas en el código a compilar.

$\langle \text{Instrucción While} \rangle ::= \text{while } (\langle \text{Condición} \rangle) \langle \text{Instrucción} \rangle$

Referencias bibliográficas

Esquivel, S. (26 de marzo de 2020). Compiladores - BNF Ejemplos [Archivo de video]. Youtube: <https://youtube.com/watch?v=lg-gDXsFxVA>

Elaine, M. (13 de Agosto de 2021). Introduction to Grammars and BNF [Archivo de video]. Youtube: <https://www.youtube.com/watch?v=f25ez8s3AsQ>

Elaborado por: César García Hernández.

Gasca Palacio Jesús Fernando

Notación BNF

La notación BNF es una notación formal que se utiliza para describir la sintaxis de un lenguaje; podemos ver este tipo de notación en lenguajes de programación.

Esta notación está compuesta de un conjunto de reglas ya anteriores establecidas que definen cómo se pueden combinar los elementos del lenguaje para poder crear declaraciones o expresiones válidas.

Para crear un lenguaje utilizando BNF, se deben seguir estos pasos:

- 1) Definir los elementos básicos del lenguaje estos pueden ser palabras clave, diferentes operadores y también los tipos de datos que se pueden utilizar.
- 2) Usando la sintaxis BNF podemos definir las reglas para combinar los elementos anteriores y así de esta manera poder crear declaraciones y expresiones.
- 3) Definir reglas y restricciones que se van a aplicar ejemplo de ello es el orden en que deben aparecer los elementos o los tipos de datos que se pueden usar.
- 4) Probar la notación BNF usando programas de muestra en el lenguaje o algún tipo de prueba (test) para verificar que sean sintácticamente correctos.
- 5) Refinar la notación BNF según sea necesario en función de los resultados de las pruebas, con la intención de mejorar y eliminar errores.

Al usar BNF, utilizaremos los siguientes símbolos:

Simbolo	Definición
$::=$ Definición	Indica que el elemento de la izquierda se puede definir según el esquema de la derecha.
Opción	Indica que puede elegirse un simbolo de los elementos separados por este.
{ } Repetición	Indica que los elementos dentro de ellos se pueden repetir Varias Veces
[] Opción	Indica que los elementos dentro de ellos pueden utilizarse o no utilizarse
() Agrupación	Indica la agrupación de los elementos incluidos en su interior

Ejemplos

$\langle \text{Identificador} \rangle ::= \langle \text{letra} \rangle \langle \text{Resto} \rangle \mid \langle \text{Guion} \rangle \langle \text{Resto} \rangle$

$\langle \text{letra} \rangle ::= a \mid b \mid c \mid \dots \mid z$

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

$\langle \text{Resto} \rangle ::= \{ \langle \text{letra} \rangle \mid \langle \text{Dígito} \rangle \mid \langle \text{Guion} \rangle \}$

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

$\langle \text{Número} \rangle ::= 0 \mid \langle \text{Uno} \rangle \langle \text{Resto} \rangle$

$\langle \text{Resto} \rangle ::= \{ \langle \text{Dígito} \rangle$

$\langle \text{Dígito} \rangle ::= 0 \mid \langle \text{Uno} \rangle$

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

• → No terminales

• → Terminales

• → Metacaracteres

BNF (BACKUS - NAUR)

La notación de Backus-Naur (BNF) es una notación formal para describir la sintaxis de los lenguajes de programación. Se basa en una serie de reglas que definen los elementos básicos del lenguaje y cómo se combinan para formar expresiones más complejas.

Las reglas de BNF se escriben en forma de tablas de símbolos, donde cada símbolo representa un elemento del lenguaje. Las reglas se indican mediante flechas que indican cómo se combinan los elementos para formar expresiones más complejas.

Por ejemplo, una sintaxis de una expresión aritmética simple: $\text{Expresión} \rightarrow \text{Term} + \text{Expresión}$ | $\text{Term} \rightarrow \text{Factor} * \text{Term}$ | $\text{Factor} \rightarrow \text{Número} \mid (\text{Expresión})$.

Esta tabla de símbolos incluye notación utilizada para describir la gramática en lenguajes formales.

Estas reglas se representan mediante símbolos no terminales, símbolos terminales y símbolos especiales como las flechas \rightarrow y el símbolo de barra vertical " $|$ ".

Christian Muriel González Moncera