Universidad Autónoma del Estado de Hidalgo

Reporte de práctica

Análisis sintáctico. Ejercicios

Catedrático:
Dr. Eduardo Cornejo-Velázquez

Alumno:
José de Jesús Sánchez Lara

1. Introducción

El análisis sintáctico es un proceso fundamental en las fases del compilador, en consecuencia, es importante para los estudiantes de ciencias computacionales y de todo aquel que deseé ingresarse en el estudio de un compilador. Por consiguiente, un analizador sintáctico tiene la tarea de indicar si una secuencia de componentes léxicos se encuentran en orden correspondiente. Para ello se utilizan diagramas, gramáticas y demás componentes y algoritmos para poder satisfacer el análisis.

2. Objetivo

El objetivo de este trabajo es poder entender la forma en que un analizador sintáctico trabaja, los elementos que requiere para hacerlo, además, se busca entender la relación entre los mismos. Finalmente, se pretende aplicar los conocimientos adquiridos a ejercicios prácticos sobre analizadores sintácticos.

3. Marco teórico

Análisis sintáctico

La tarea principal de un analizador sintáctico es indicar si la secuencia de componentes léxicos, encontrados en el análisis léxico, están en orden correspondiente a las reglas gramaticales del lenguaje.
Un analizador sintáctico recibe como entrada los componentes léxicos y da como resultado un árbol sintáctico o árbol de análisis sintáctico. [1]

Gramática ambigua

Se define como una gramática ambigua aquella gramática capaz de generar más de un árbol sintáctico para una misma secuencia de componentes. [1]

Gramáticas

Es una especificación para la estructura sintáctica de un lenguaje de programación. [1]

Beneficios de las gramáticas

- Una gramática define la sintaxis de un lenguaje de programación de forma clara y comprensible.
- Permite construir analizadores sintácticos eficientes para determinar la estructura de un programa.
- Facilita la traducción del código fuente a código objeto y la detección de errores.
- Posibilita la evolución del lenguaje al agregar nuevas construcciones de manera estructurada.

Componentes de una gramática libre de contexto

- Conjunto de símbolos terminales: elementos básicos del lenguaje definido por la gramática.
- Conjunto de símbolos no terminales: representan agrupaciones de terminales y conducen a más producciones.
- Conjunto de producciones: reglas con un no terminal en el lado izquierdo y una secuencia de terminales y no terminales en el derecho.
- Símbolo inicial: un no terminal designado como punto de inicio de la gramática.

Árboles de análisis sintáctico

Son árboles etiquetados en cuanto a una derivación mostrando gráficamente el símbolo inicial de una gramática. [1]

4. Herramientas empleadas

Las herramientas utilizadas para poder llevar a cabo la práctica son un creador de diagramas, una herramienta de edición y redacción del enlace, por último, la fuente de los ejercicios, el libro.

- **Overleaf:** Overleaf es una plataforma en línea para la escritura, de documentos LaTeX. Facilita la gestión de proyectos científicos y académicos.
- **Creatly:** Creatly es una herramienta para crear diagramas en línea que permite la creación de mapas conceptuales, diagramas UML, diagramas de flujo y otros esquemas gráficos.
- **Libro de referencia:** Este libro contiene los ejercicios que se resolverán en la actividad, proporcionando una base teórica y práctica para su desarrollo.

5. Desarrollo de la práctica

Ejercicio 1

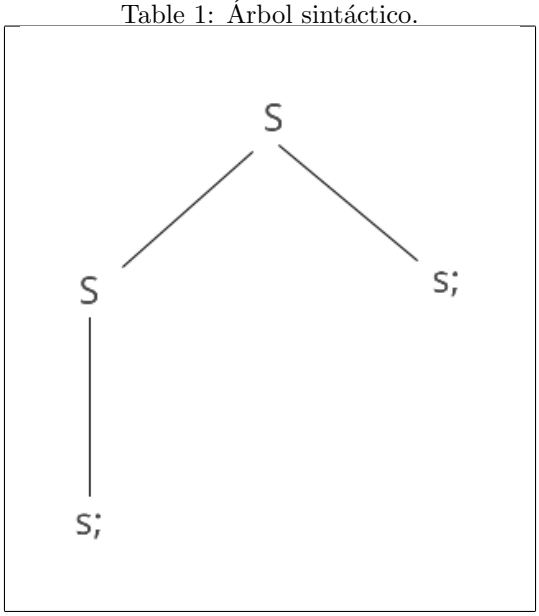
5.1.1 a) Escriba una gramática que genere el conjunto de cadenas {s;, s;s;, s;s;s;, ...}

Una gramática formal se define como $G = (N, \Sigma, P, S)$

- N es el conjunto de símbolos no terminales.
- Σ es el conjunto de símbolos terminales.
- P es el conjunto de reglas de producción.
- S es el símbolo inicial.
- Símbolos no terminales: $N = \{S\}$.
- Símbolos terminales: $\Sigma = \{s;\}$.
- Reglas de producción: P contiene las siguientes reglas:
  - $S \rightarrow S\, s;$
  - $S \rightarrow s;$
- Símbolo inicial: S.

5.1.2 b) Genere un árbol sintáctico para la cadena s;s;

Table 1: Árbol sintáctico.



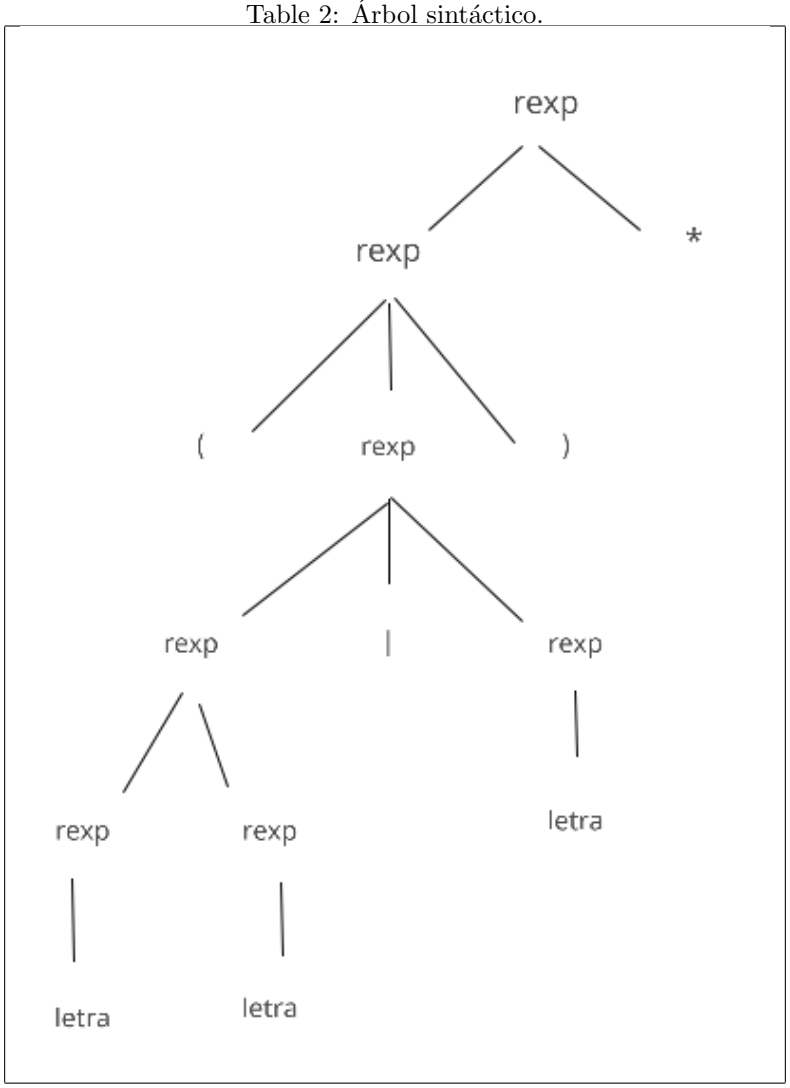
Ejercicio 2

5.2.1 Considere la siguiente gramática:

- $rexp \rightarrow rexp$“ | ”$rexp$
- | $rexp\, rexp$
- | $rexp$ “ $*$ ”
- | “(” $rexp$ “)”
- | $letra$

5.2.2 a) Genere un árbol sintáctico para la siguiente expresión regular $(ab \mid b)^*$

Table 2: Árbol sintáctico.



Ejercicio 3

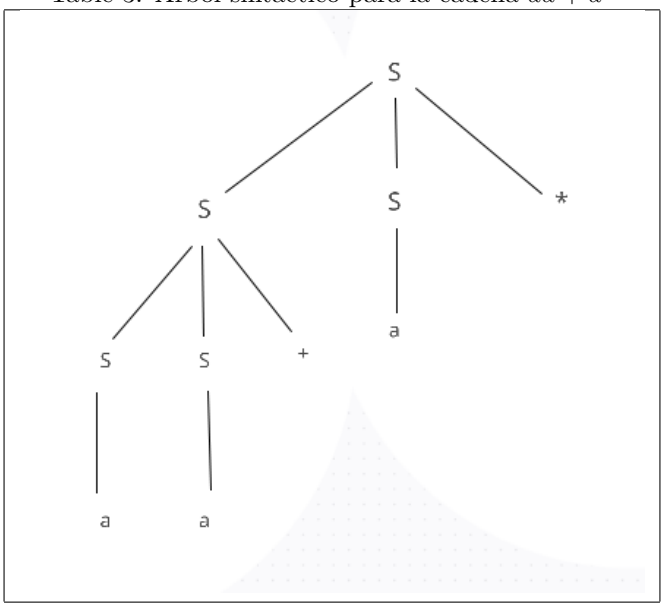
5.3.1 De las siguientes gramáticas, describa el lenguaje generado por la gramática y genere árboles sintácticos con las respectivas cadenas.

- a) $S \rightarrow SS+ \mid SS^* \mid a$ con la cadena $aa + a^*$
- b) $S \rightarrow 0S1 \mid 01$ con la cadena 000111
- c) $S \rightarrow +SS \mid^* SS \mid a$ con la cadena $+^*aaa$

5.3.2 a)

- 1. De forma general, es capaz de expresar una suma de cadenas, una multiplicación de cadenas, una cadena por sí sola o una combinación de operaciones con cadenas donde los operadores son la suma y multiplicación.
- 2. Ejemplo: {a, aa+, aa*, aa+aa*+, aa+a+, ... }
- 3. Árbol sintáctico para la cadena $aa + a^*$

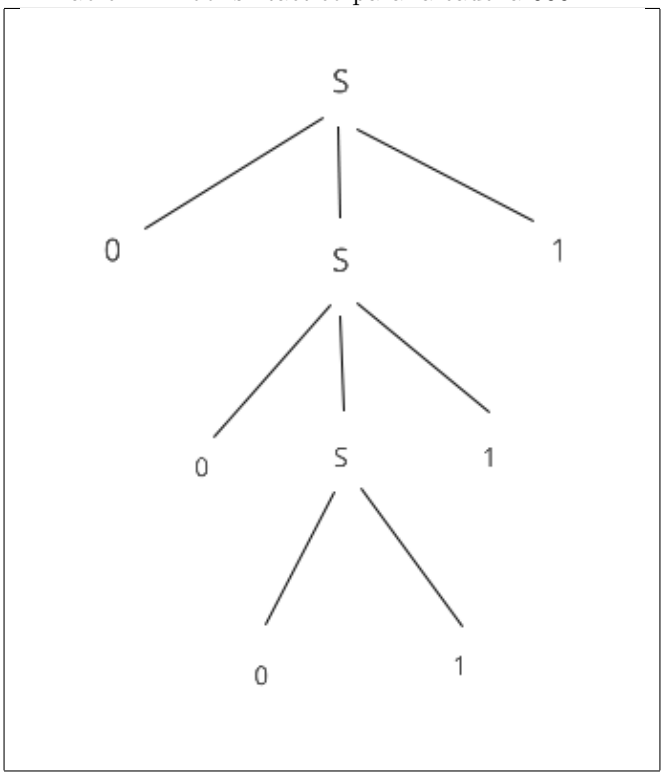
Table 3: Árbol sintáctico para la cadena $aa + a^*$



5.3.3 b)

- 1. Es capaz de generar una cadena que contiene un número n de 0s al inicio y un número m de 1s al final. La variable m es igual a la variable n.
- 2. Ejemplo: {01, 0011, 000111, 00001111, ... }
- 3. Árbol sintáctico para la cadena 000111

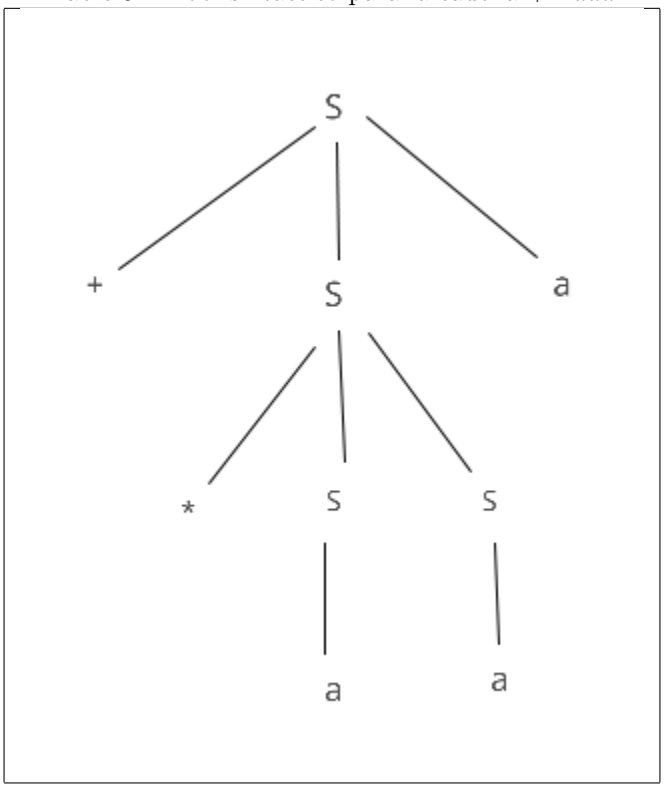
Table 4: Árbol sintáctico para la cadena 000111



5.3.4 c)

- 1. De forma general, es capaz de expresar una suma de cadenas, una multiplicación de cadenas y una cadena sola, sin embargo, se puede observar que si se utiliza el primer y segundo elemento de la producción, entonces, siempre iniciará la cadena con un operador.
- 2. Ejemplo: {a, +aa, *aa, +a+aa, ... }
- 3. Árbol sintáctico para la cadena $+ * aaa$

Table 5: Árbol sintáctico para la cadena $+ * aaa$



Ejercicio 4

5.4.1 ¿Cuál es el lenguaje generado por la siguiente gramática:

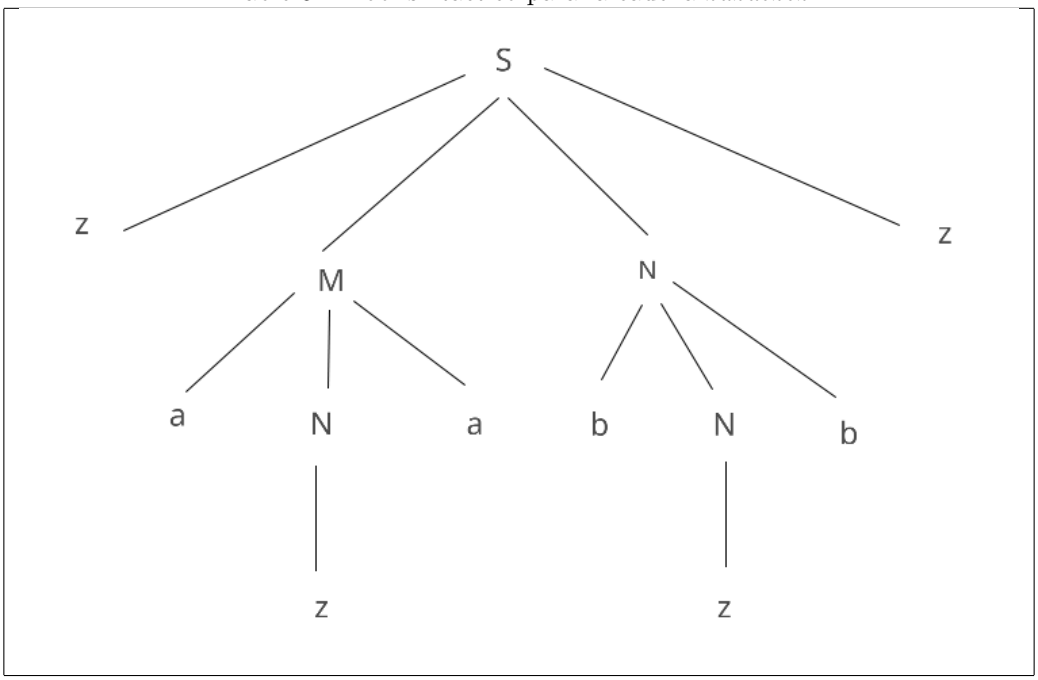
- a) $S \rightarrow xSy \mid \epsilon$
- La gramática puede generar el lenguaje $L : \{\epsilon, xy, xxyy, ...\}$
- Esa gramática genera un lenguaje donde se puede tener la cadena vacía o una cadena conformada por un número n de x's y un número m de y´s, donde m es igual a n.

Ejercicio 5

5.5.1 Genere el árbol sintáctico para la cadena zazabzbz utilizando la siguiente gramática:

- a) $S \rightarrow zMNz$
- b) $M \rightarrow aNa$
- c) $N \rightarrow bNb$
- d) $N \rightarrow z$

Table 6: Árbol sintáctico para la cadena *zazabzbz*



Ejercicio 6

5.6.1 Demuestre que la gramática que se presenta a continuación es ambigua, mostrando que la cadena ictictses tiene derivaciones que producen distintos árboles de análisis sintáctico.

- a) $S \rightarrow ictS$
- b) $M \rightarrow ictSeS$
- c) $N \rightarrow s$
- La forma de comprobarlo es a través de las derivaciones de la siguiente manera:
- A) Para el árbol 1 se siguen las siguientes derivaciones y reglas de producción:
  - 1) $S \rightarrow ictS$
  - 2) $ictS \rightarrow ictictSeS$
  - 3) $ictS \rightarrow ictictSeS$
  - 4) $ictictSeS \rightarrow ictictseS$
  - 5) $ictictseS \rightarrow ictictses$
- B) Para el árbol 2 se siguen las siguientes derivaciones y reglas de producción:
  - 1) $S \rightarrow ictSeS$
  - 2) $ictSeS \rightarrow ictictSeS$
  - 3) $ictictSeS \rightarrow ictictseS$
  - 4) $ictictseS \rightarrow ictictses$

Table 7: Primer árbol

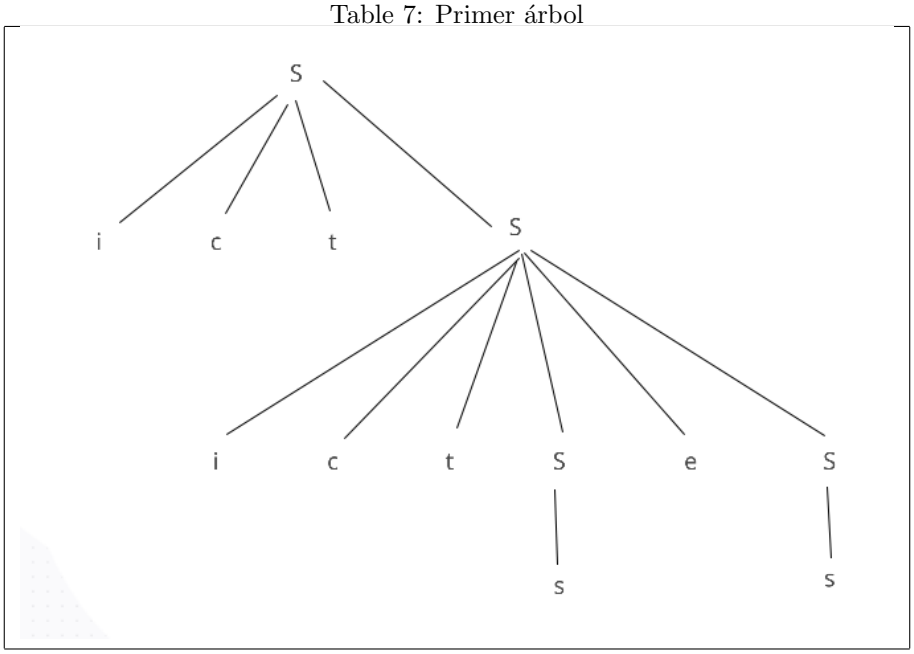
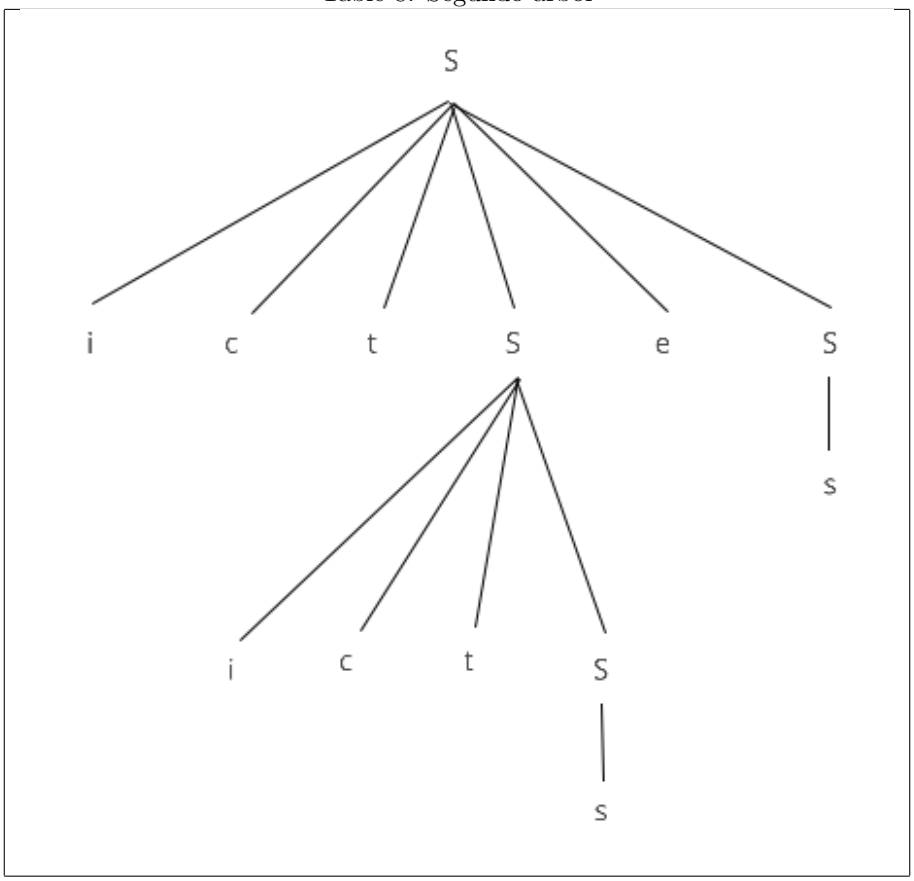


Table 8: Segundo árbol



Ejercicio 7

5.7.1 Considere la siguiente gramática

- a) $S \rightarrow (L) \mid a$
- b) $L \rightarrow L, S \mid S$

5.7.2 Encuéntrense árboles de análisis sintáctico para las siguientes frases:

- a) (a, a)
- b) $(a, (a, a))$
- c) $(a, ((a, a), (a, a)))$

Table 9: Primer árbol

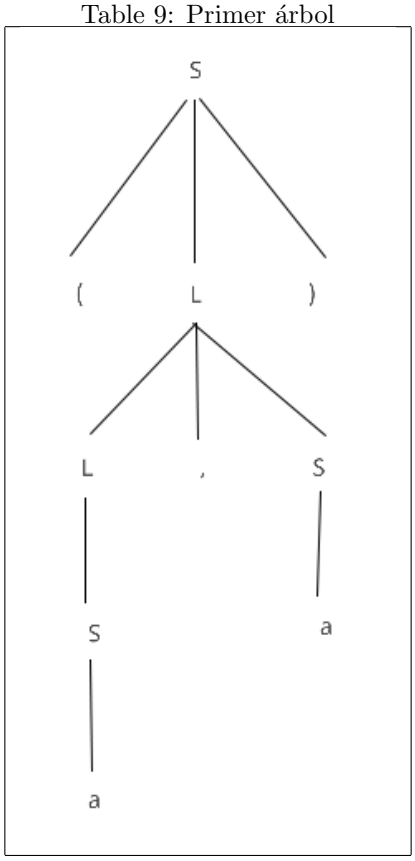


Table 10: Segundo árbol

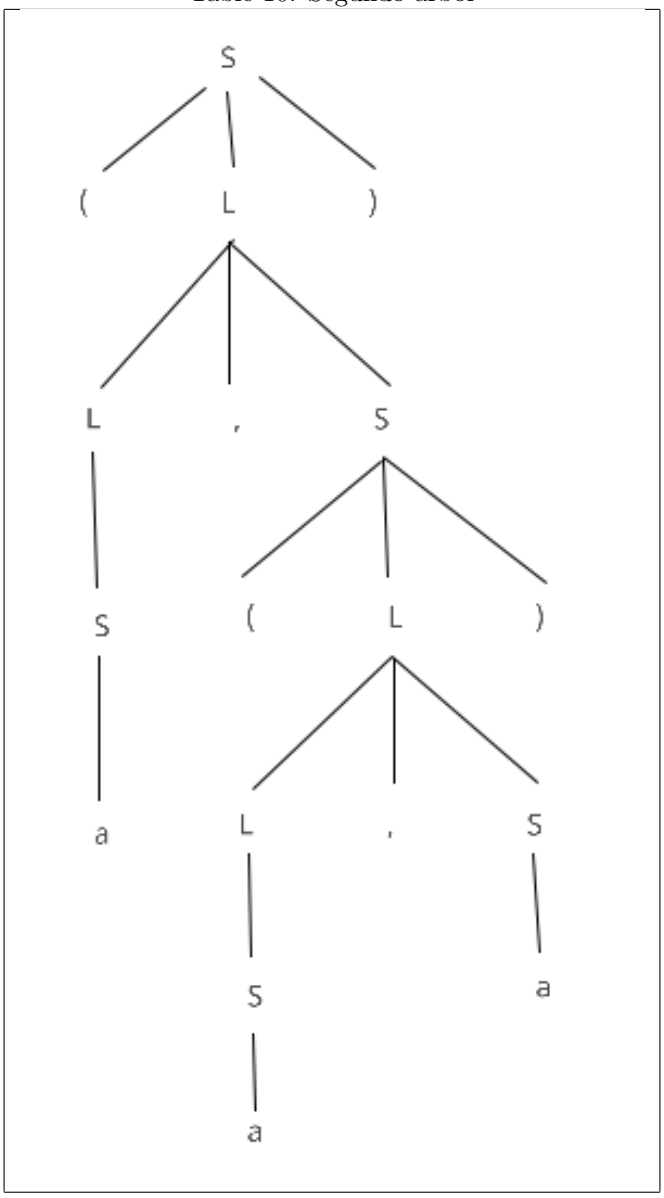
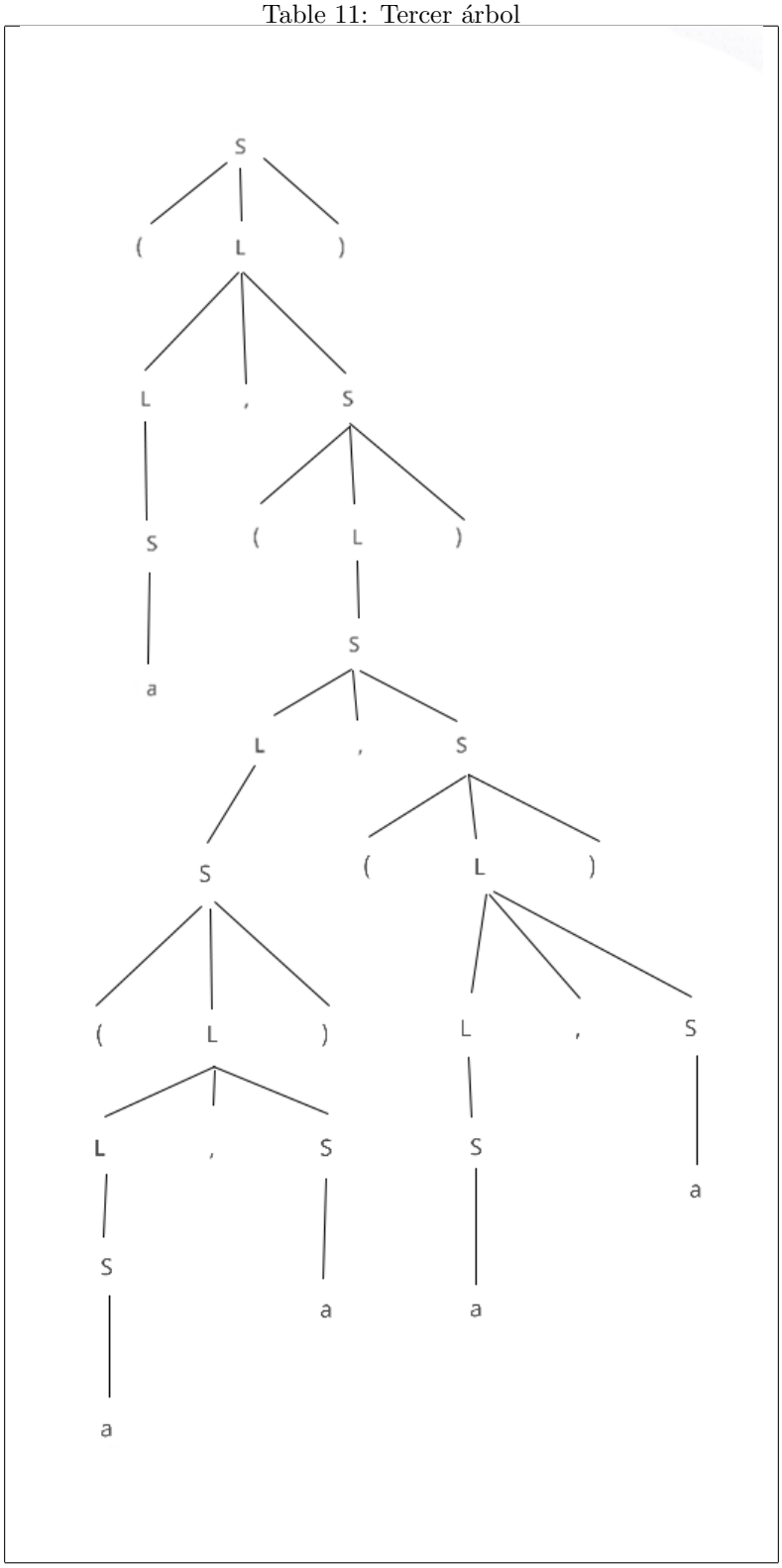


Table 11: Tercer árbol

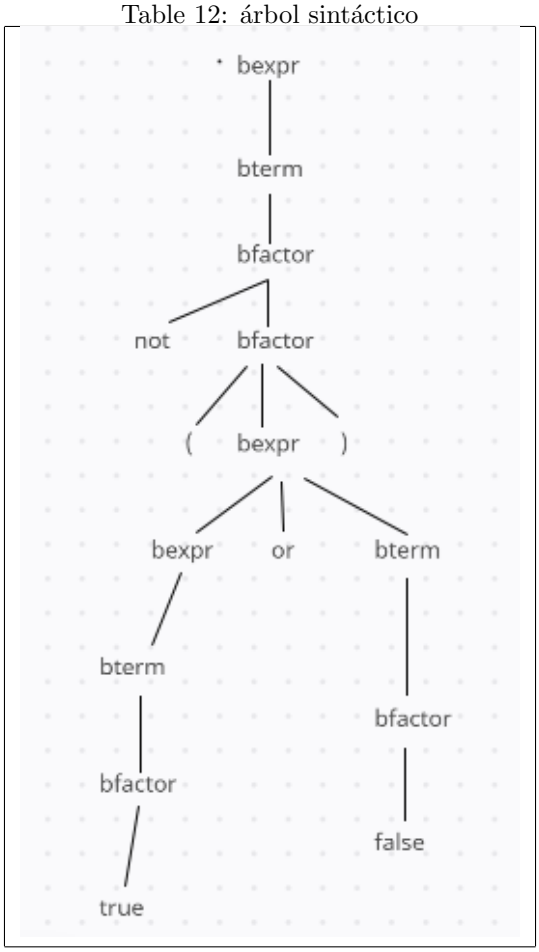


Ejercicio 8

5.8.1 Constrúyase un árbol sintáctico para la frase not (true or false) y la gramática:

- a) $bexpr \rightarrow bexpr$ or $bterm \mid bterm$
- b) $bterm \rightarrow bterm$ and $bfactor \mid bfactor$
- c) $bfactor \rightarrow$ not $bfactor \mid (bexpr) \mid$ true $\mid$ false

Table 12: árbol sintáctico



Ejercicio 9

5.9.1 Diseñe una gramática para el lenguaje del conjunto de todas las cadenas de símbolos 0 y 1 tales que todo 0 va inmediatamente seguido de al menos un 1.

- Símbolos no terminales: $N = \{S\}$.
- Símbolos terminales: $\Sigma = \{0, 1\}$.
- Reglas de producción: $\{S- > SS, S- > 1S, S- > 01S, S- > \epsilon\}$
- Símbolo inicial: S.

5.9.2 Gramática generada:

- $S \to SS$
- $S \to 1S$
- $S \to 01S$
- $S \to \epsilon$

Ejercicio 10

5.10.1 Elimine la recursividad por la izquierda de la siguiente gramática

- a) $S \to (L) \mid a$
- b) $L \to L, S \mid S$

5.10.2 Procedimiento

1. Analizar que partes de la gramática contienen recursividad
   - a) $S \to (L) \mid a$
   - **b)** $L \to L, S \mid S - >$ **Es la regla de producción con recursividad**
2. Se toma la segunda regla de producción y se reescribe siguiente manera
   - $L \to L, S \mid S$
   - $L \to SL'$
   - $L' \to, SL' \mid \epsilon$
3. Ahora sustituimos la regla de producción recursiva por las nuevas reglas. La nueva gramática obtiene la siguiente forma:
   - $S \to (L) \mid a$
   - $L \to sL'$
   - $L' \to, SL' \mid \epsilon$

Se puede observar que la recursividad se ve eliminada puesto que ya no existe una regla de la forma A $\to$ Aa

Ejercicio 11

5.11.1 Dada la gramática $S \to (S) \mid x$, escriba un pseudocódigo para el análisis sintáctico de esta gramática mediante el método descendente recursivo.

5.11.2 Pseudocódigo

Función $S()$
Elegir una producción $S \to X_1X_2...X_k$
Para cada X_i en $X_1X_2...X_k$ **hacer**:
Si X_i es un **no terminal entonces Llamar** a la función $X_i()$
Sino Si X_i es un **terminal entonces Comparar** con el carácter actual en la entrada **Si** coincide, **avanzar** al siguiente símbolo **Sino**, **mostrar** "Error de sintaxis"

Ejercicio 12

5.12.1 Qué movimientos realiza un analizador sintáctico predictivo con la entrada (id+id)*id, mediante el algoritmo 3.2, y utilizándose la tabla de análisis sintáctico de la tabla 3.1. (Tómese como ejemplo la Figura 3.13)

Pila	Entrada	Acción
$E	(id + id)*id$	$E \to TE'$
$E'T	(id + id)*id$	$T \to FT'$
$E'T'F	(id + id)*id$	$F \to (E)$
$E'T')E(	(id + id)*id$	"(" concuerda
$E'T')E	id + id)*id$	$E \to TE'$
$E'T')E'T	id + id)*id$	$T \to FT'$
$E'T')E'T'F	id + id)*id$	$F \to id$
$E'T')E'T'id	id + id)*id$	"id" concuerda
$E'T')E'T'	+ id)*id$	$T \to \epsilon$
$E'T')E'	+ id)*id$	$E' \to +TE'$
$E'T')E'T+	+ id)*id$	"+" concuerda
$E'T')E'T	id)*id$	$T \to FT'$
$E'T')E'T'F	id)*id$	$F \to id$
$E'T')E'T'id	id)*id$	"id" concuerda
$E'T')E'T'	)*id$	$T' \to \epsilon$
$E'T')E'	)*id$	$E' \to \epsilon$
$E'T')	)*id$	")" coincide
$E'T'	*id$	$T' \to *FT'$
$E'T'F*	*id$	"*" coincide
$E'T'F	id$	$F \to id$
$E'T'id	id$	id coincide
$E'T'	$	$T' \to \epsilon$
$E'	$	$E' \to \epsilon$
$	$	aceptar()

Table 13: Tabla

No terminal	Símbolo de entrada					
	id	+	*	(	)	$
E	E → TE ‘			E → TE ‘		
E ‘		E ‘→ +TE ‘			E ‘ → ε	E ‘ → ε
T	T → FT ‘			T → FT ‘		
T ‘		T ‘ → ε	T ‘→ *FT ‘		T ‘ → ε	T ‘ → ε
F	*F* → **id**			*F* → (*E*)		

Table 14: Análisis

Algoritmo 3.2: Análisis sintáctico predictivo, controlado por una tabla. (Aho, Lam, Sethi, & Ullman, 2008, pág. 226)

Entrada: Una cadena *w* y una tabla de análisis sintáctico *M* para la gramática *G*.

Salida: Si *w* está en el lenguaje de la gramática *L*(*G*), una derivación por la izquierda de *w*; de lo contrario, una indicación de error.

Método: Al principio, el analizador sintáctico se encuentra en una configuración con *w*$ en el búfer de entrada, y el símbolo inicial *S* de *G* en la parte superior de la pila, por encima de $.

establecer *ip* para que apunte al primer símbolo de *w*;
establecer *X* con el símbolo de la parte superior de la pila;
while (*X* ≠ $) { /* mientras la pila no está vacía*/
    if (*X* es *a*) extraer de la pila y avanzar *ip*; /* *a*=símbolo al que apunta *ip* */
    else if (*X* es un terminal) *error*( )
        else if (*M*[*X*, *a*] es una entrada de error) *error*( )
            else if ($M[X, a] = X \rightarrow Y_1 Y_2 \ldots Y_k$) {
                enviar de salida la producción $X \rightarrow Y_1 Y_2 \ldots Y_k$;
                extraer de la pila;
                meter $Y_k, Y_{k-1}, \ldots, Y_1$ en la pila, con Y_1 en la parte superior;
                }
    establecer X con el símbolo de la cima de la pila;
}

Ejercicio 13

5.13.1 La gramática 3.2, sólo maneja las operaciones de suma y multiplicación, modifique esa gramática para que acepte, también, la resta y la división; Posteriormente, elimine la recursividad por la izquierda de la gramática completa y agregue la opción de que F, también pueda derivar en num, es decir, $F \to (E) \mid id \mid num$

Regla 3.2:

- **Gramática:**
  - $E \to E + T \mid T$
  - $T \to T * F \mid F$
  - $F \to (E) \mid \textbf{id}$

Gramática ampliada con la resta y la división, con recursividad

- **Gramática:**
  - $E \to E + T \mid E - T \mid T$
  - $T \to T * F \mid T/F \mid F$
  - $F \to (E) \mid \textbf{id} \mid num$

Eliminación de la recursividad de las regla $E \to E + T \mid E - T \mid T$ y la regla $T \to T * F \mid T/F \mid F$
Para la regla: $E \to E + T \mid E - T \mid T$

- **Gramática generada:**
  - $E \to E + T \mid E - T \mid T$
  - $E \to TE'$
  - $E' \to +TE' \mid -TE \mid \epsilon$

Para la regla:$T \to T * F \mid T/F \mid F$

- **Gramática generada:**
  - $T \to T * F \mid T/F \mid F$
  - $T \to FT'$
  - $T' \to *FT' \mid /FT' \mid \epsilon$

Resultado

- $E \to TE'$
- $E' \to +TE' \mid -TE \mid \epsilon$
- $T \to FT'$
- $T' \to *FT' \mid /FT' \mid \epsilon$
- $F \to (E) \mid \textbf{id} \mid num$

Ejercicio 14

5.14.1 Escriba un pseudocódigo (e implemente en Java) utilizando el método descendente recursivo para la gramática resultante del ejercicio anterior (ejercicio 13).

a)Pseudocódigo:

Definir una lista llamada componentes
Definir una lista llamada conversion
Definir una variable complex como de tipo String
Definir una variable "i" como entero, inicializada en 0 (variable de control)

INICIO PROGRAMA

FUNCIÓN E():

Definir *error* como entero y asignarle 0
error = T()
SI *error* es 0 o 1 ENTONCES *error* = Ep() FIN SI
RETORNAR error

FIN FUNCIÓN

FUNCIÓN Ep():

Definir *error* como entero y asignarle 0
complex = componentes[i]
SI complex es suma o resta ENTONCES Incrementar i
error = T() SI *error* es 0 o 1 ENTONCES
error = Ep()
FIN SI
FIN SI
SI complex es un espacio vacío ENTONCES RETORNAR *error*
SINO
error = 1
FIN SI
RETORNAR *error*

FIN FUNCIÓN

FUNCIÓN T():

Definir *error* como entero y asignarle 0
error = F()
SI *error* es 0 ENTONCES
error = Tp()
FIN SI
RETORNAR *error*

FIN FUNCIÓN

FUNCIÓN Tp():
Definir *error* como entero y asignarle 0
complex = componentes[i]
SI complex es multiplicacion o division ENTONCES
Incrementar i
error = F()
SI error es 0 ENTONCES
error = Tp()
FIN SI
FIN SI
SI complex es un espacio vacío ENTONCES
RETORNAR *error*
SINO
error = 1
FIN SI
RETORNAR *error*

FIN FUNCIÓN

FUNCIÓN F():
Definir *error* como entero y asignarle 0
complex = componentes[i]
SI complex es (ENTONCES
Incrementar i
error = E()
SI *error* es 0 o 1 ENTONCES
complex = componentes[i]
SI complex es) ENTONCES
error = 0
Incrementar i
SINO
error = 1
Decrementar i
FIN SI
FIN SI
SINO SI complex es id o num ENTONCES
Incrementar i
SINO
error = 2
Decrementar i
FIN SI
RETORNAR *error*

FIN FUNCIÓN
FIN Programa Recursivo

b)Implementación

Table 15: Parte 1

```
import java.util.Scanner;

public class Programa_recursivo {

    public static String[] componentes;
    public static String[] conversion;
    public static String complex;
    public static int i = 0;

    public static int E() {

        int error = 0;
        error = T();
        if (error == 0 || error == 1) {
            error = Ep();
        }

        return error;
    }

    public static int Ep() {
        int error = 0;
        complex = componentes[i];

        if (complex.equals("suma") || complex.equals("resta")) {
            i++;
            error = T();
            if (error == 0 || error == 1) {
                error = Ep();
```

Table 16: Parte 2

```
            }
        }
        if (complex.equals(" ")) {
            return error;
        } else {
            error = 1;
        }
        return error;
    }

    public static int T() {
        int error = 0;
        error = F();
        if (error == 0) {
            error = Tp();
        }

        return error;
    }

    public static int Tp() {
        int error = 0;
        complex = componentes[i];

        if (complex.equals("multiplicacion") || complex.equals("division")) {
            i++;
```

Table 17: Parte 3

```
            error = F();
            if (error == 0) {
                error = Tp();
            }
        }
        if (complex.equals(" ")) {
            return error;
        } else {
            error = 1;
        }
        return error;
    }

    public static int F() {
        int error = 0;
        complex = componentes[i];

        if (complex.equals("(")) {
            i++;
            error = E();

            if (error == 0 || error == 1) {
                complex = componentes[i];
                if (complex.equals(")")) {
                    error = 0;
                    i++;
                } else {
                    error = 1;
```

Table 18: Parte 4

```
                    i--;
                }
            }
        } else {

            if (complex.equals("id") || complex.equals("num")) {
                i++;
            } else {
                error = 2;
                i--;
            }
        }

        return error;
    }

    public static void main(String[] args) {
        int error;

        Scanner scanner = new Scanner(System.in);
        System.out.print("Ingresa el numero de componentes: ");
        int numeroComponentes = scanner.nextInt();
        scanner.nextLine();

        componentes = new String[numeroComponentes + 1];
        conversion = new String[numeroComponentes + 1];

        componentes[numeroComponentes] = " ";

        for (int j = 1; j <= numeroComponentes; j++) {
```

Table 19: Parte 5

```
        for (int j = 1; j <= numeroComponentes; j++) {
            System.out.print("Ingresa el componente " + j + ": ");
            String texto = scanner.nextLine();
            conversion[j - 1] = texto;

            if (texto.matches("\\d+")) {
                componentes[j - 1] = "num";
            } else if (texto.matches("[a-zA-Z][a-zA-Z0-9]*")) {
                componentes[j - 1] = "id";
            } else if (texto.matches("\\(")) {
                componentes[j - 1] = "(";
            } else if (texto.matches("\\)")) {
                componentes[j - 1] = ")";
            } else if (texto.matches("\\+")) {
                componentes[j - 1] = "suma";
            } else if (texto.matches("-")) {
                componentes[j - 1] = "resta";
            } else if (texto.matches("\\*")) {
                componentes[j - 1] = "multiplicacion";
            } else if (texto.matches("/")) {
                componentes[j - 1] = "division";
            } else {
                componentes[j - 1] = "u";
            }
        }
        error = E();
```

Table 20: Parte 6

```
        if (error == 0) {
            System.out.println("\nCadena aceptada de la forma: ");
            for (int i = 0; i < numeroComponentes; i++) {
                System.out.print(componentes[i] + " ");
            }
            System.out.println("\n Cadena equivalente a: ");
            for (int i = 0; i < numeroComponentes; i++) {
                System.out.print(conversion[i] + " ");
            }
        } else {
            System.out.println("\nCadena rechazada: ");
            for (int i = 0; i < numeroComponentes; i++) {
                System.out.print(conversion[i] + " ");
            }
        }

    }
}
```

b)Resultado

Table 21: Implementación

```
Ingresa el numero de componentes: 13
Ingresa el componente 1: (
Ingresa el componente 2: 1
Ingresa el componente 3: +
Ingresa el componente 4: 2
Ingresa el componente 5: +
Ingresa el componente 6: (
Ingresa el componente 7: x
Ingresa el componente 8: *
Ingresa el componente 9: y
Ingresa el componente 10: )
Ingresa el componente 11: /
Ingresa el componente 12: z
Ingresa el componente 13: )

Cadena aceptada de la forma:
( num suma num suma ( id multiplicacion id ) division id )
 Cadena equivalente a:
( 1 + 2 + ( x * y ) / z ) BUILD SUCCESSFUL (total time: 25 seconds)
```

5. Conclusiones

Un analizador sintáctico es una herramienta fundamental, ya que permite analizar y comprender el orden en que llegan los tokens proporcionados por el analizador léxico. Se logró comprender cómo las gramáticas son esenciales para establecer las rutinas y la estructura que debe seguir el analizador sintáctico.

En consiguiente, los árboles sintácticos me fueron de gran utilidad, ya que proporcionaron una representación gráfica de cómo se puede derivar una cadena a partir de una gramática. Dichos árboles pueden ser ambiguos o no ambiguos; son ambiguos cuando la gramática es capaz de generar más de un árbol sintáctico para una misma cadena. Sin embargo, es posible eliminar la recursividad a través de ciertos métodos.

Además, existen diferentes formas de construir o recorrer un árbol, ya sea desde las hojas hacia la raíz o viceversa. Cuando se nos da una gramática, suele ser más sencillo construir el árbol desde la raíz, aunque también es posible hacerlo desde las hojas, siempre y cuando no exista un ciclo infinito.

Finalmente, el programa desarrollado fue capaz de representar una gramática y determinar la validez de una cadena. No obstante, se presentaron dificultades al intentar representar la cadena vacía.

Referencias Bibliográficas

References

[1] Carranza Sahagún, D. U. (**2024**). *Compiladores: Fase de análisis*. Editorial TransDigital. ISBN: 978-607-26754-0-7.