Universidad Autónoma del Estado de Hidalgo
Instituto de Ciencias Básicas e Ingeniería
Licenciatura en Ciencias Computacionales

2.4 Análisis sintáctico

Ejercicios

Autómatas y Compiladores
Dr. Eduardo Cornejo-Velázquez
Semestre 6 Grupo 3
Alumna Ashley Torres Perez

Ejercicio 1

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

La secuencia de cadenas sugiere una gramática recursiva, la cual puede ser definida como una gramática libre de contexto (GLC). Una GLC es una cuádrupla $G = (N, T, P, S)$ donde:

- **Conjunto de símbolos no terminales** (N) : Representan estructuras sintácticas intermedias.
- **Conjunto de símbolos terminales** (T) : Son los símbolos finales de la cadena (lo que realmente aparece en el lenguaje).
- **Conjunto de reglas de producción** (P) : Definen cómo se pueden transformar los símbolos no terminales en terminales o en otros no terminales.
- **Símbolo inicial** (S) : Es el punto de partida de la derivación de las cadenas del lenguaje.

Según Carraza (2024), una gramática recursiva por la izquierda es aquella donde el primer símbolo del lado derecho de una producción es el mismo símbolo no terminal del lado izquierdo de la producción. En otras palabras tiene la forma $A \to A\alpha$. Para la secuencia $\{s;, s; s;, s; s; s;, ...\}$ se puede escribir una gramática recursiva por la izquierda de la siguiente manera:

$$A \to s$$

$$B \to ;$$

$$S \to AB$$

$$S \to SAB$$

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

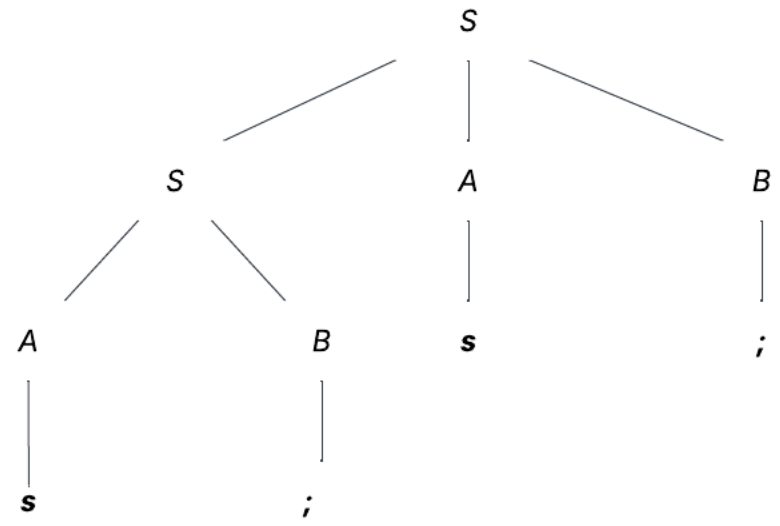

Figure 1: árbol sintáctico para la cadena $s; s;$

Ejercicio 2

Considere la siguiente gramática:

rexp → *rexp* "|" *rexp*
| *rexp rexp*
| *rexp* "*"
| "(" *rexp* ")"
| **letra**

Figure 2: gramática del ejercicio 2

a) Genere un árbol sintáctico para la expresión regular $(ab|b)*$.

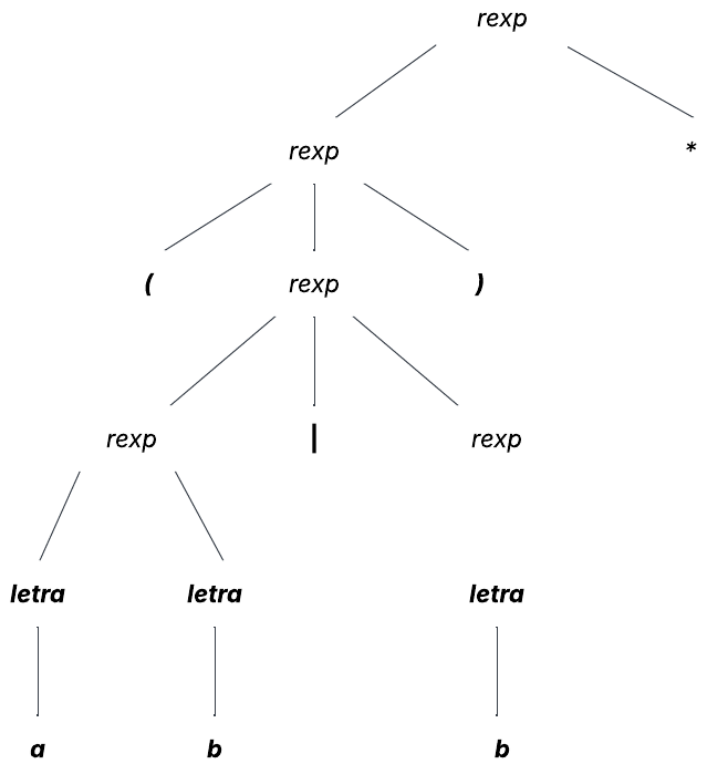

Figure 3: gramática del ejercicio 2

Ejercicio 3

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

a) $S \to SS+ \mid SS* \mid a$ con la cadena $aa + a*$.

b) $S \to 0S1 \mid 01$ con la cadena 000111.

c) $S \rightarrow +SS \mid *SS \mid a$ con la cadena $*aaa$.

a. $S \rightarrow SS+ \mid SS* \mid a$ con la cadena $aa+a*$

La gramática genera expresiones formadas por varias "a" combinadas con los operadores + y *.

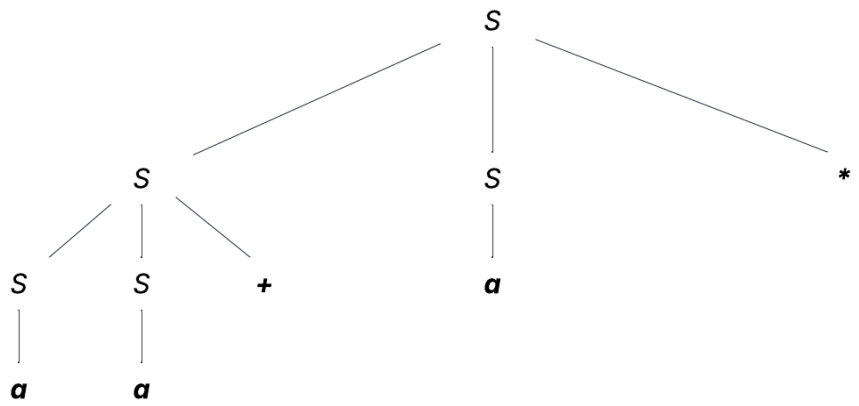

Figure 4: árbol sintáctico para gramática a

b. $S \rightarrow 0S1 \mid 01$ con la cadena 000111

Esta gramática genera cadenas balanceadas de ceros y unos donde cada '0' tiene un '1' correspondiente. Siempre hay un 0 inicial y un 1 final que se emparejan correctamente.

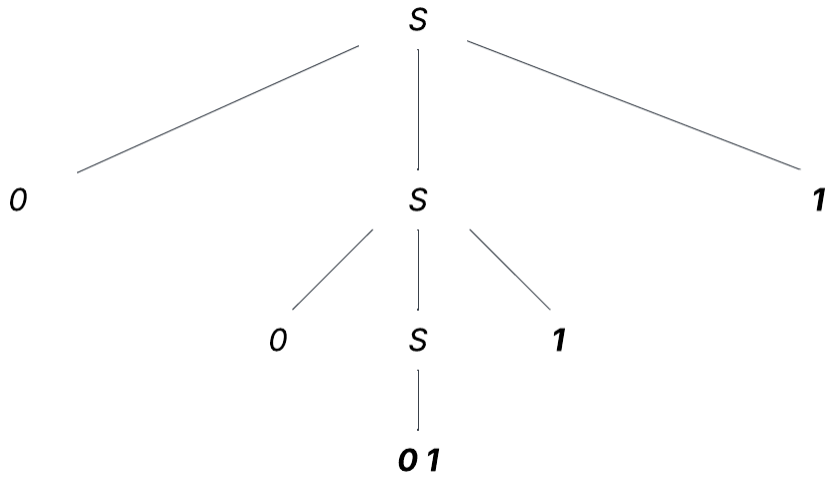

Figure 5: árbol sintáctico para gramática b

c. $S \rightarrow +SS \mid *SS \mid a$ con la cadena $+*aaa$

Esta gramática genera expresiones en notación prefija (Polaca).

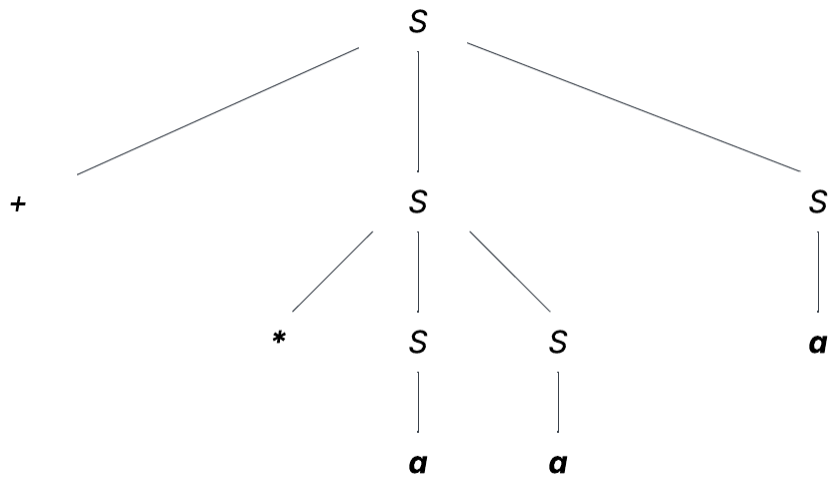

Figure 6: árbol sintáctico para gramática c

Ejercicio 4

¿Cuál es el lenguaje generado por la siguiente gramática? Dada la gramática:

$$S \to xSy \mid \varepsilon$$

La producción $S \to xSy$ indica que cada vez que expandimos S, agregamos una x al inicio y una y al final. La producción $S \to \varepsilon$ permite detener la recursión, generando la cadena vacía.

Lenguaje generado:
La gramática genera cadenas con la misma cantidad de x y y, siempre en el formato:

$$\begin{aligned} \varepsilon &\quad \text{(cadena vacía)} \\ xy & \\ xxyy & \\ xxxyyy & \\ xxxxyyyy & \\ \vdots & \end{aligned}$$

El patrón general de las cadenas generadas es:

$$x^n y^n, \quad n \geq 0$$

El lenguaje generado por la gramática es:

$$L = \{x^n y^n \mid n \geq 0\}$$

Ejercicio 5

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

$$S \to zMNz$$

$$M \to aNa$$

$$N \to bNb$$

$$N \to z$$

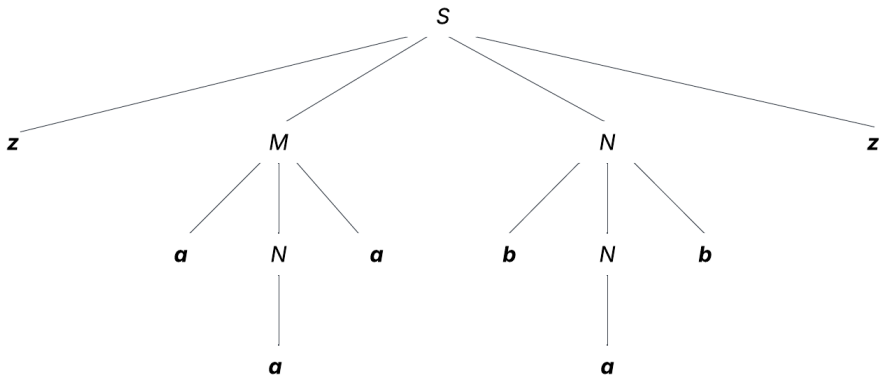

Figure 7: árbol sintáctico para la cadena zazabzbz

Ejercicio 6

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.

$$S \to ictS$$

$$S \to ictSeS$$

$$S \to s$$

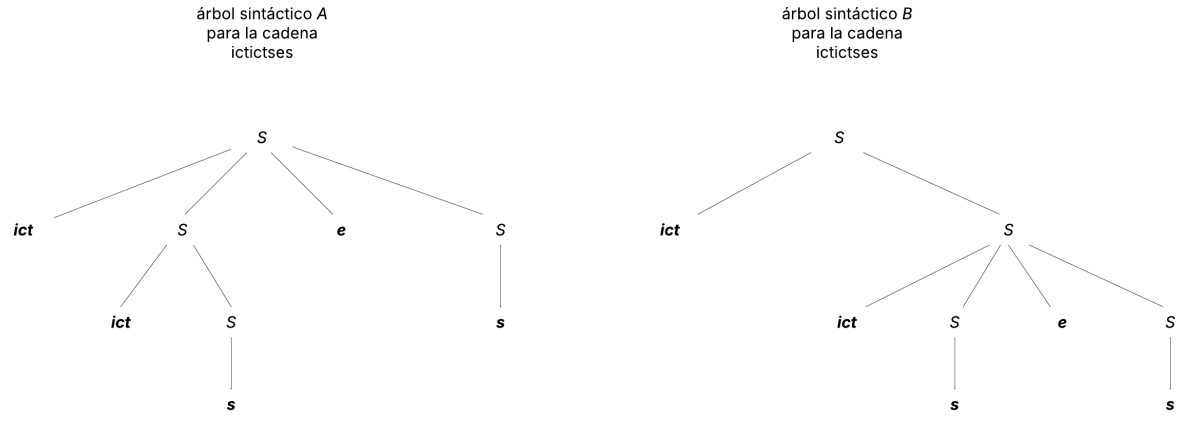

Figure 8: árbol sintáctico para la cadena zazabzbz

Ejercicio 7

Considere la siguiente gramática:

$$S \to (L)|a$$
$$L \to L, S|S$$

Encuéntre los árboles de análisis sintáctico para las siguientes frases:

a) (a, a).

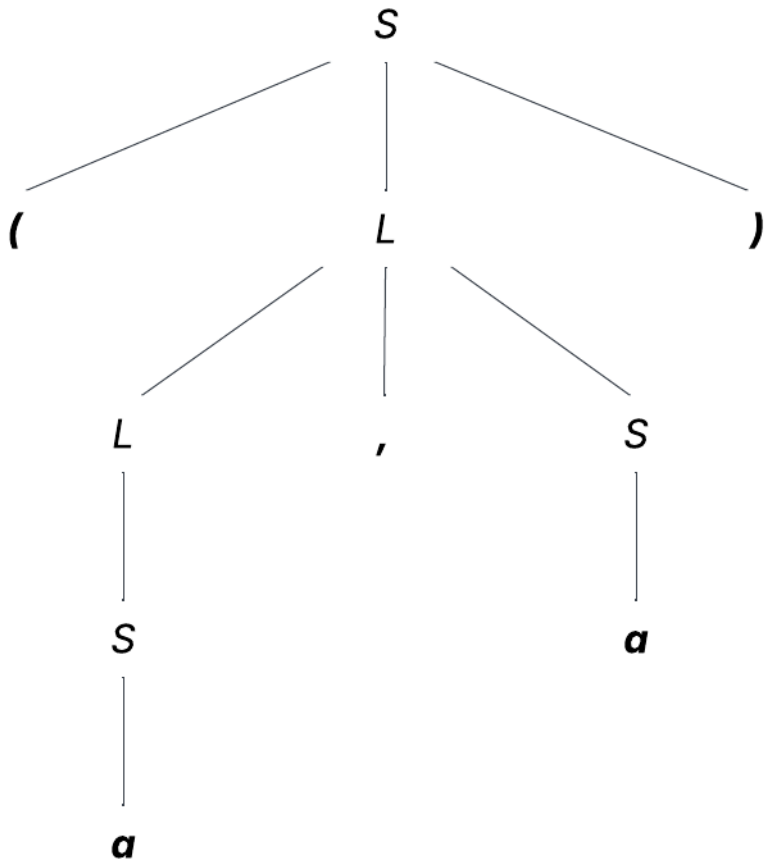

Figure 9: árbol sintáctico para la cadena (a, a)

b) $(a, (a, a))$.

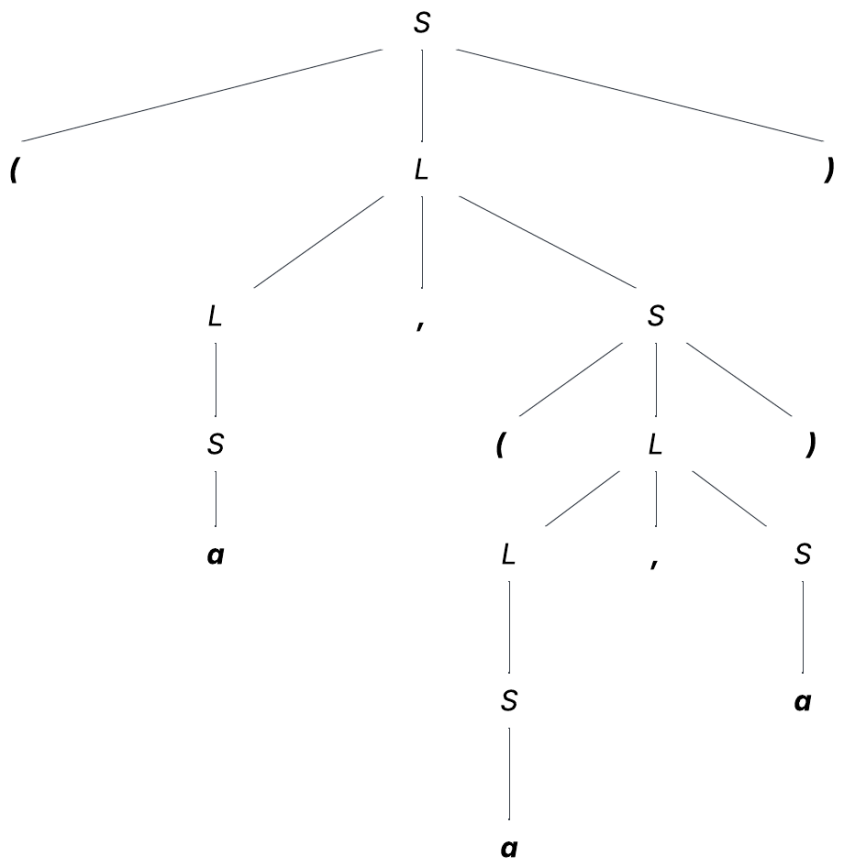

Figure 10: árbol sintáctico para la cadena $(a, (a, a))$

c) $(a, ((a, a), (a, a)))$.

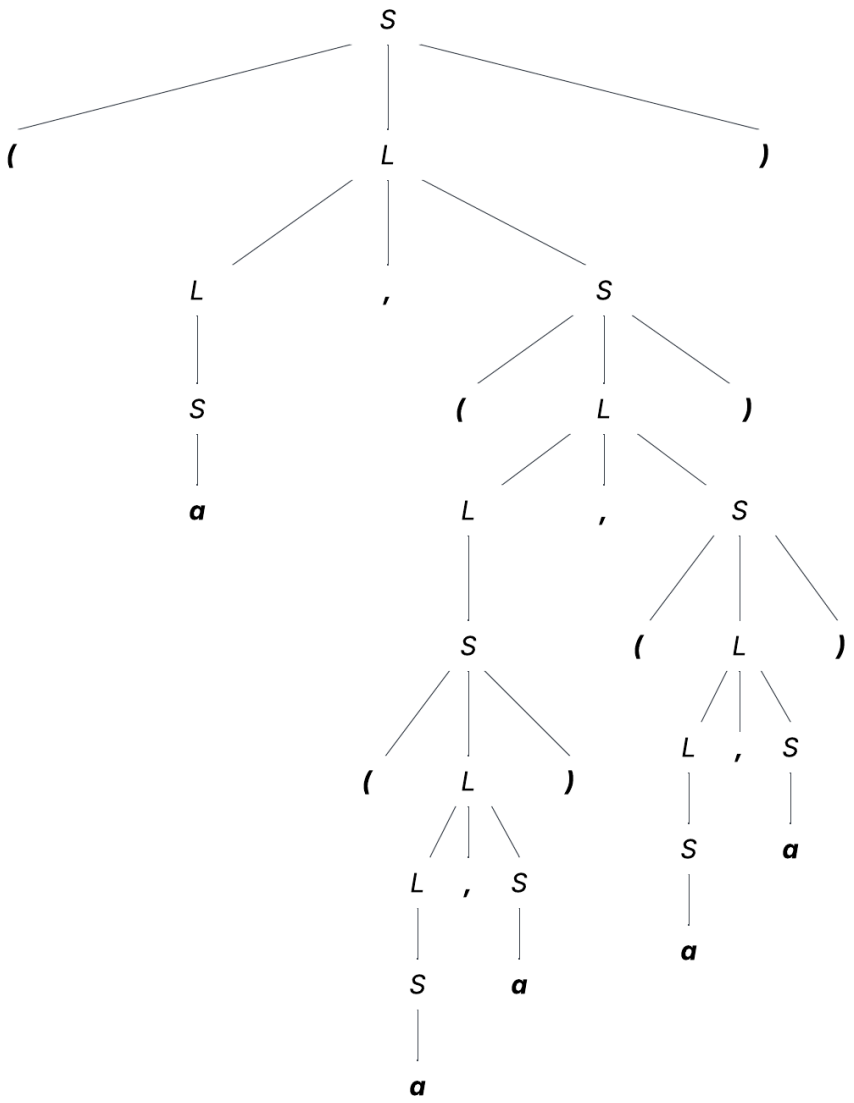

Figure 11: árbol sintáctico para la cadena $(a, ((a, a), (a, a)))$

Ejercicio 8

Constrúyase un árbol sintáctico para la frase $not(trueorfalse)$ y la gramática:

$$\begin{aligned} \text{bexpr} &\rightarrow \text{bexpr or bterm} \mid \text{bterm}, \\ \text{bterm} &\rightarrow \text{bterm and bfactor} \mid \text{bfactor}, \\ \text{bfactor} &\rightarrow \text{not bfactor} \mid \text{(bexpr)} \mid \text{true} \mid \text{false}. \end{aligned}$$

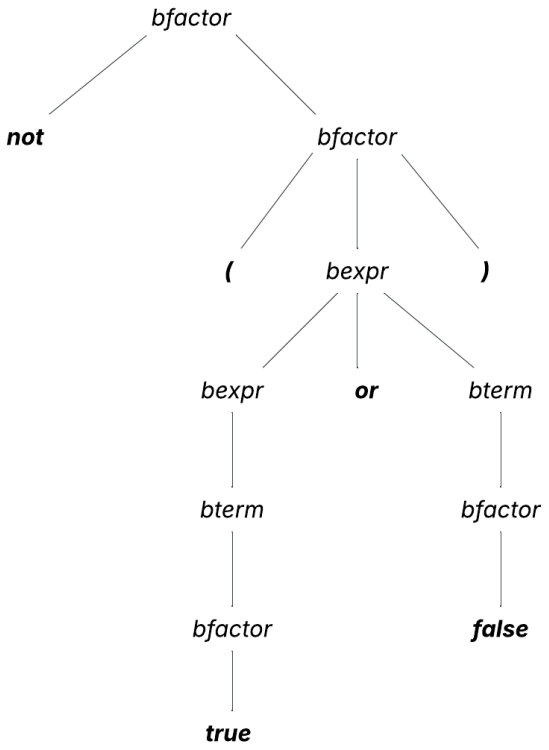

Figure 12: árbol sintáctico para la frase $not(trueorfalse)$

Ejercicio 9

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.

$$S \to A|B|\epsilon$$

$$A \to 0B$$

$$B \to 1S$$

Esta gramática define un lenguaje que genera cadenas formadas por los símbolos 0 y 1 tales que todo 0 va inmediatamente seguido de al menos un 1, incluyendo la cadena vacía.
El símbolo S es el símbolo inicial y puede derivar en A, B o en la cadena vacía. Si se elige A, la producción indica que debe comenzar con un 0 seguido de B, lo que significa que cualquier cadena derivada desde A siempre empezará con un 0. Si se elige B, la producción indica que debe comenzar con un 1 seguido de S, lo que implica que cualquier derivación desde B inicia con un 1 y puede continuar con más símbolos generados a partir de S.
Dado que S también puede producir la cadena vacía, esto permite que la gramática genere combinaciones de 0 y 1 en cierto orden, así como la cadena vacía.

Ejercicio 10

Elimine la recursividad por la izquierda de la siguiente gramática:

$$S \to (L)|a$$

$$L \to L, S|S$$

$$S \to (L) \mid a$$
$$L \to SL'$$
$$L' \to, SL' \mid \varepsilon$$

Ejercicio 11

Dada la gramática $Sß(S)|x$, escriba un pseudocódigo para el análisis sintáctico de esta gramática mediante el método descendente recursivo.

```
funcion analizar_expresi n() {
    si token_actual es '(' entonces
        consumir_token('(')
        analizar_expresion()
        consumir_token(')')
    sino si token_actual es 'x' entonces
        consumir_token('x')
    sino
        lanzar_error("Token inesperado")
    fin si
}
```

Ejercicio 12

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).
El analizador:

1. Empieza con E en la pila
2. Expande E → TE'
3. Cuando ve '(', expande T → FT'
4. Continúa hasta consumir toda la entrada

Pasos que toma el analizador:

Pila	Entrada	Acción
$E	(id+id)*id$	E→TE'
$E'T	(id+id)*id$	T→FT'

Ejercicio 13

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 → (E) — id — num

Gramática 3.2

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

Gramática Modificada

$$E \to TE'$$
$$E' \to +TE' \mid -TE' \mid \varepsilon$$
$$T \to FT'$$
$$T' \to *FT' \mid /FT' \mid \varepsilon$$
$$F \to (E) \mid id \mid num$$

Ejercicio 14

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

```
1  public class Parser {
2      private Token tokenActual;
3      private final AnalizadorLexico analizadorLexico;
4
5      public Parser(AnalizadorLexico analizadorLexico) {
6          this.analizadorLexico = analizadorLexico;
7          this.tokenActual = analizadorLexico.obtenerSiguienteToken();
8      }
9
10     public void analizar() throws ErrorSintactico {
11         procesarExpresion();
12         if (tokenActual.tipo != TipoToken.FIN) {
13             lanzarError("Caracteres␣inesperados␣despu s␣de␣la␣expresi n");
14         }
15     }
16
17     private void procesarExpresion() throws ErrorSintactico {
18         procesarTermino();
```

```
19          procesarExpresionRecursiva();
20      }
21
22      private void procesarExpresionRecursiva() throws ErrorSintactico {
23          if (esOperadorSumaResta()) {
24              TipoToken operador = tokenActual.tipo;
25              consumirToken();
26              procesarTermino();
27              registrarOperacion(operador);
28              procesarExpresionRecursiva();
29          }
30      }
31
32      private void procesarTermino() throws ErrorSintactico {
33          procesarFactor();
34          procesarTerminoRecursivo();
35      }
36
37      private void procesarTerminoRecursivo() throws ErrorSintactico {
38          if (esOperadorMultiplicacionDivision()) {
39              TipoToken operador = tokenActual.tipo;
40              consumirToken();
41              procesarFactor();
42              registrarOperacion(operador);
43              procesarTerminoRecursivo();
44          }
45      }
46
47      private void procesarFactor() throws ErrorSintactico {
48          switch (tokenActual.tipo) {
49              case PARENTESIS_IZQUIERDO:
50                  consumirToken(TipoToken.PARENTESIS_IZQUIERDO);
51                  procesarExpresion();
52                  consumirToken(TipoToken.PARENTESIS_DERECHO);
53                  break;
54              case NUMERO_ENTERO:
55              case NUMERO_DECIMAL:
56              case IDENTIFICADOR:
57                  registrarOperando(tokenActual);
58                  consumirToken();
59                  break;
60              default:
61                  lanzarError("Factor␣no␣v lido");
62          }
63      }
64
65      private boolean esOperadorSumaResta() {
66          return tokenActual.tipo == TipoToken.SUMA ||
67                 tokenActual.tipo == TipoToken.RESTA;
68      }
69
70      private boolean esOperadorMultiplicacionDivision() {
71          return tokenActual.tipo == TipoToken.MULTIPLICACION ||
72                 tokenActual.tipo == TipoToken.DIVISION ||
73                 tokenActual.tipo == TipoToken.MODULO;
74      }
75
76      private void consumirToken() throws ErrorSintactico {
77          tokenActual = analizadorLexico.obtenerSiguienteToken();
78      }
79
80      private void consumirToken(TipoToken tipoEsperado) throws ErrorSintactico {
81          if (tokenActual.tipo != tipoEsperado) {
82              lanzarError("Se␣esperaba:␣" + tipoEsperado);
83          }
84          consumirToken();
85      }
86
```

```
87      private void registrarOperacion(TipoToken operador) {
88          // Implementaci n para registrar la operaci n en el AST
89      }
90
91      private void registrarOperando(Token operando) {
92          // Implementaci n para registrar operandos
93      }
94
95      private void lanzarError(String mensaje) throws ErrorSintactico {
96          throw new ErrorSintactico(mensaje + "␣en␣l nea␣" + tokenActual.linea);
97      }
98  }
99
100 enum TipoToken {
101     FIN, IDENTIFICADOR, NUMERO_ENTERO, NUMERO_DECIMAL,
102     SUMA, RESTA, MULTIPLICACION, DIVISION, MODULO,
103     PARENTESIS_IZQUIERDO, PARENTESIS_DERECHO
104 }
105
106 class Token {
107     TipoToken tipo;
108     String valor;
109     int linea;
110 }
111
112 class ErrorSintactico extends Exception {
113     public ErrorSintactico(String mensaje) {
114         super(mensaje);
115     }
116 }
```

Referencias Bibliográficas

References

[1] Carranza Sahagún, D.U. (2024). *Compiladores: Fases de análisis.* Transdigital. https://doi.org/10.56162/transdigitalb44