

# Reporte de Ejercicios 2.4

## Autómatas y Compiladores

Universidad Autónoma del Estado de Hidalgo

Instituto de Ciencias Básicas e Ingeniería

Eduardo Cornejo Velázquez

Ordaz Rangel David

Fecha: March 27, 2025



## 1 Ejercicio 1

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

1. Reglas de producción de la gramática.

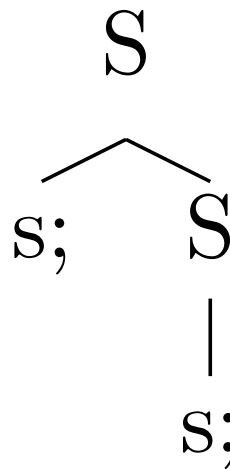
- $V$  (variables/no terminales):  $\{S\}$
- $\Sigma$  (terminales):  $\{s, ;\}$
- $P$  (producciones):

$$S \rightarrow s;$$

$$S \rightarrow s;S$$

- Símbolo inicial:  $S$

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



## 2 Ejercicio 2

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

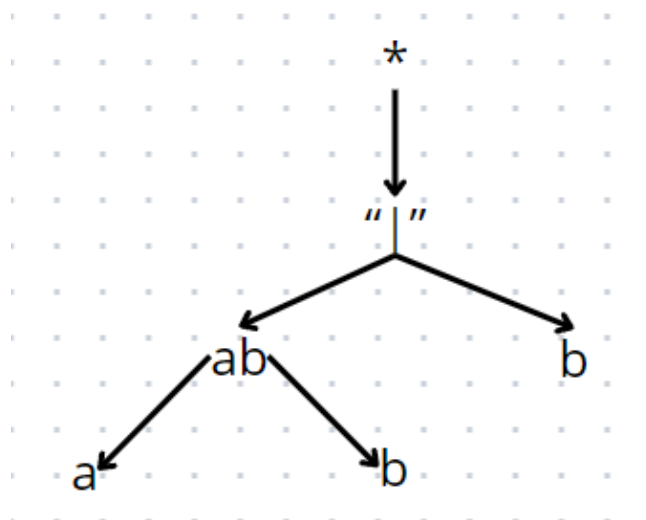


Figure 1: Árbol sintáctico de la expresión  $(ab|b)^*$ .

### 3 Ejercicio 3

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  $aa + a^*$ .

**Descripción del lenguaje:** Esta gramática solo utiliza la letra  $a$  como operando y utiliza las operaciones  $+$  y  $*$ , además es una estructura binaria

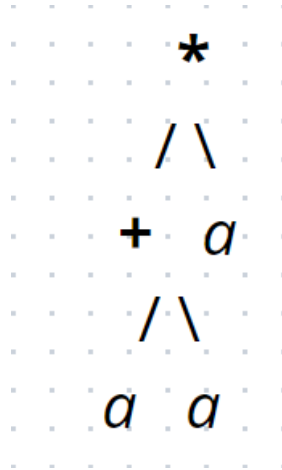


Figure 2: Árbol sintáctico de  $aa + a^*$

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

**Descripción del lenguaje:**

Genera cadenas en las que cada 0 tiene un 1 correspondiente esto permite que estén balanceadas, además cada cadena comienza con 0 y termina con 1.

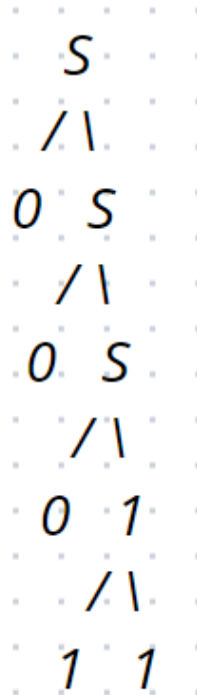


Figure 3: Árbol sintáctico de  $000111$

c)  $c)S\mathbb{B} + SS| * SS|a$  con la cadena  $+ * aaa$ .

**Descripción del lenguaje:**

Representa una notación fija en el que aparecen primero los operadores  $(+ *)$  en orden y después los operandos  $(a)$ , la estructura representa operaciones de manera muy interesante.

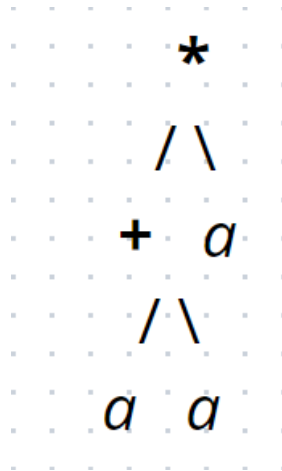


Figure 4: Árbol sintáctico de  $+ * aaa$

## 4 Ejercicio 4

¿Cuál es el lenguaje generado por la siguiente gramática?  $S\mathbb{B}xSy|e$

El lenguaje generado por la gramática:  $S \rightarrow xSy$  o  $S \rightarrow \varepsilon$  es:

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

Donde  $n$  es un número entero no negativo, lo que significa que la cantidad de 'x' debe ser igual a la cantidad de 'y' en cada cadena generada.

## 5 Ejercicio 5

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

$$\begin{aligned} S &\rightarrow zMNz \\ M &\rightarrow aNa \\ N &\rightarrow bNb \\ N &\rightarrow z \end{aligned} \tag{1}$$

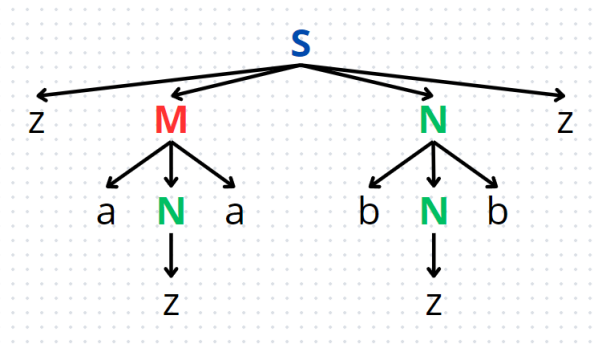


Figure 5: Árbol sintáctico de  $+ * aaa$

## 6 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.

$$\begin{aligned} S &\rightarrow ictS \\ S &\rightarrow ictSeS \\ S &\rightarrow s \end{aligned} \quad (2)$$

Para probar que la gramática es ambigua crearemos 2 árboles que expresen la misma cadena pero que sean distintos entre sí, la cadena a probar es *ictictses*

Árbol versión 1:

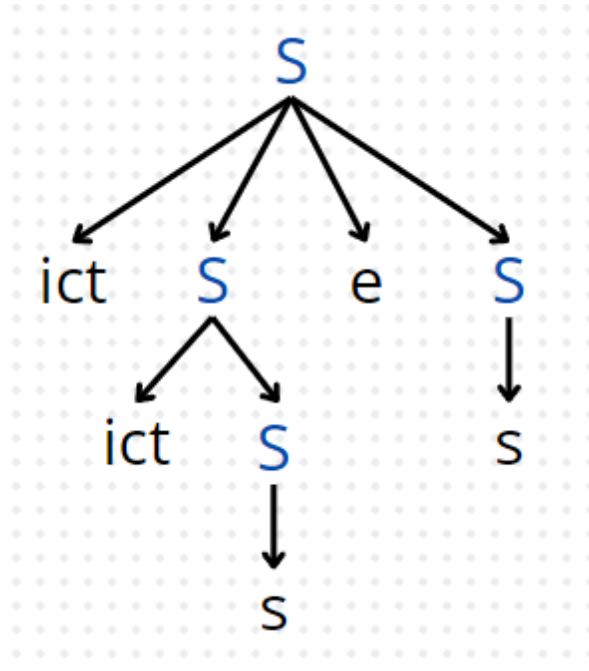


Figure 6: Árbol sintáctico v1 *ictictses*

Árbol versión 2:

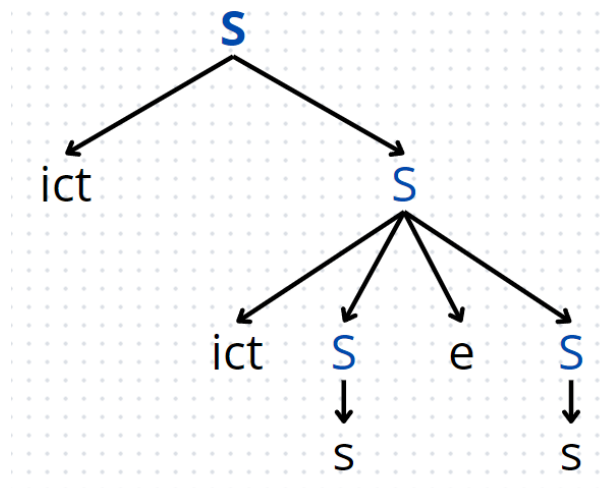


Figure 7: Árbol sintáctico v2 *ictictses*

Ergo verum est

## 7 Ejercicio 7

Considere la siguiente gramática

$$\begin{aligned} S &\rightarrow (L)|a \\ L &\rightarrow L,S|S \end{aligned} \quad (3)$$

Encuéntrense árboles de análisis sintáctico para las siguientes frases:

a)  $(a, a)$

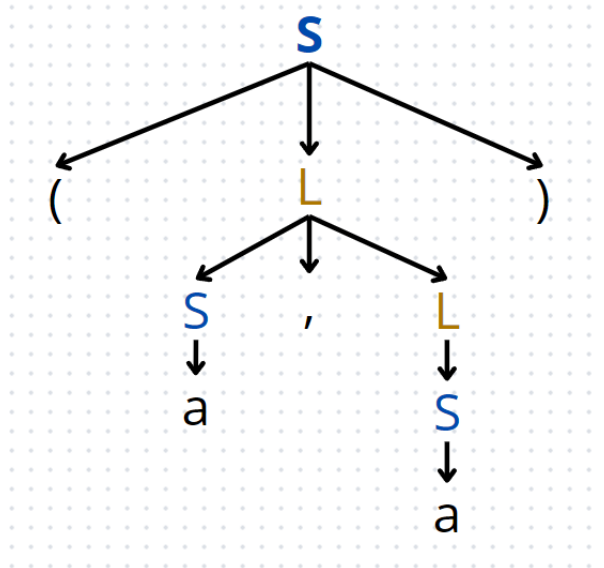


Figure 8: Árbol de  $(a, a)$

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

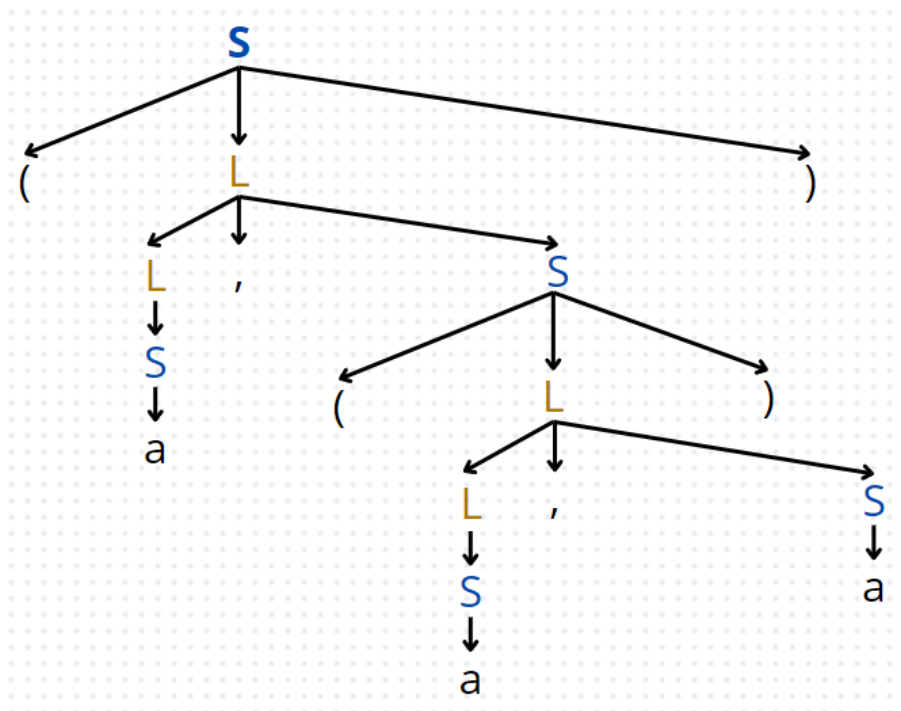


Figure 9: Árbol de  $(a, (a, a))$

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

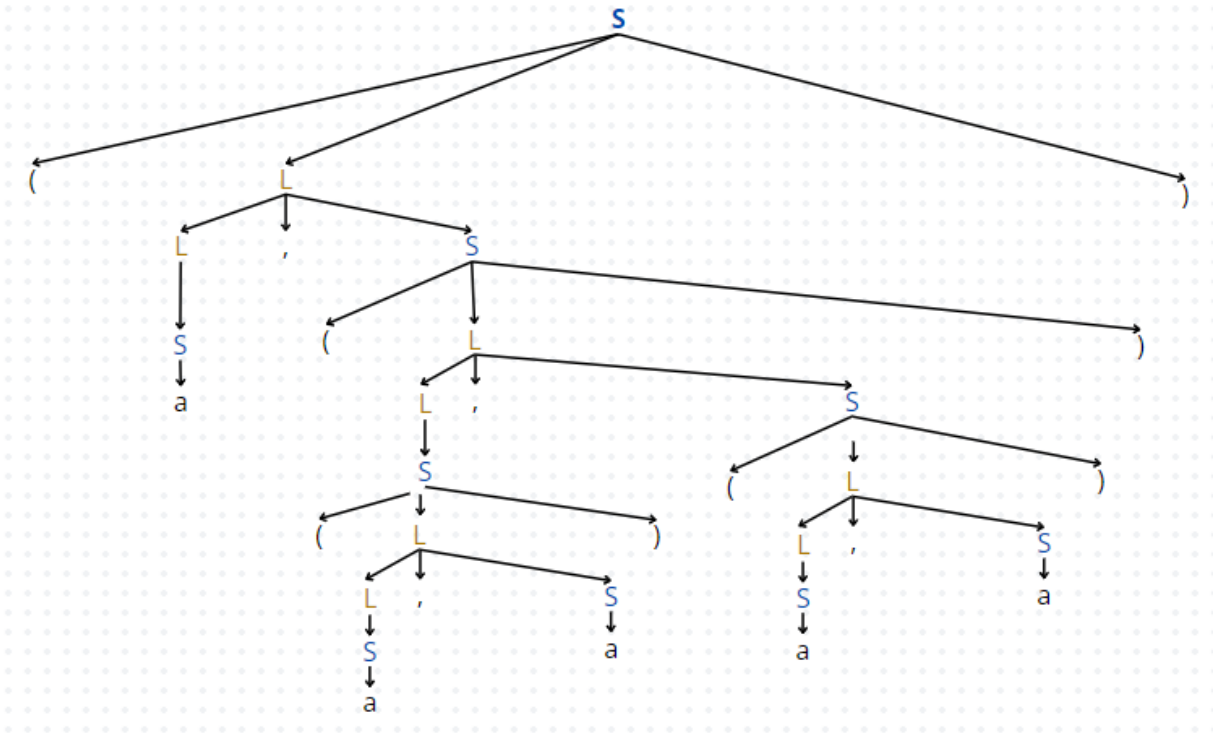


Figure 10: Árbol de  $(a, ((a, a), (a, a)))$

## 8 Ejercicio 8

Considere la siguiente gramática

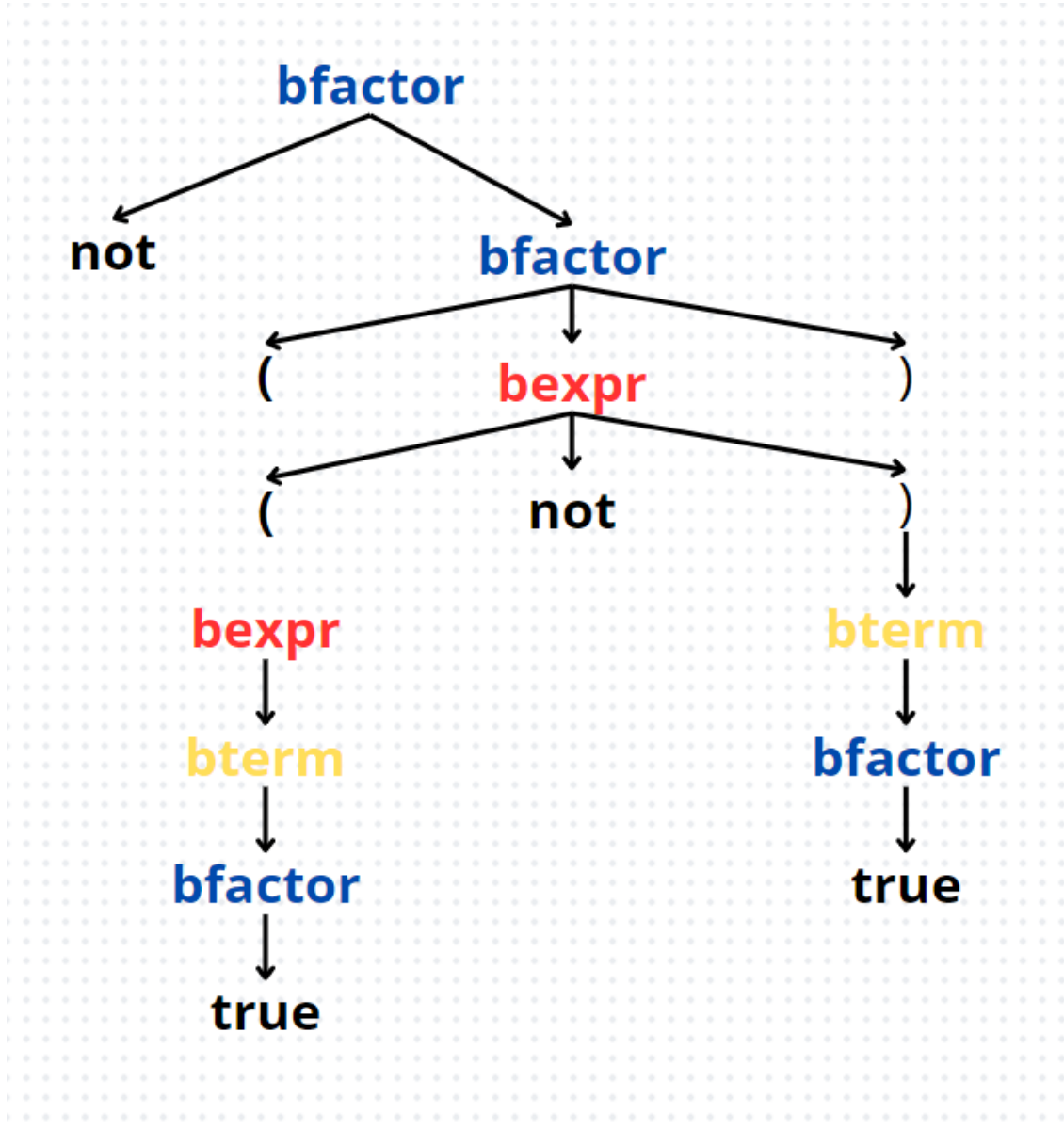
$$\begin{aligned} \text{bexpr} &\rightarrow \text{bexpr or bterm} \mid \text{bterm} \\ \text{bterm} &\rightarrow \text{bterm and bfactor} \mid \text{bfactor} \\ \text{bfactor} &\rightarrow \text{not bfactor} \mid (\text{bexpr}) \mid \text{true} \mid \text{false} \end{aligned}$$


Figure 11: Árbol de la gramática 8



## 9 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.

La gramática está definida por los siguientes componentes:

- **Variables:**  $S$  (símbolo inicial).
- **Alfabeto:**  $\{0, 1\}$ .
- **Reglas de producción:**

$$S \rightarrow 1S \mid 01S \mid \epsilon$$

**Explicación de las reglas:**

- $S \rightarrow 1S$ : Permite cadenas que comienzan con un 1 y luego siguen cualquier cadena que cumpla con la condición.
- $S \rightarrow 01S$ : Asegura que cada 0 va seguido inmediatamente de un 1 (como lo requiere la condición), y luego se puede seguir con cualquier cadena que cumpla la misma condición.
- $S \rightarrow \epsilon$ : Permite que la cadena esté vacía, ya que una cadena vacía también cumple con la condición de no tener 0 sin un 1 que lo siga.

**Ejemplos de cadenas válidas:**

- 111
- 011
- 010101
- 101

**Cadenas no válidas:**

- 001 (el primer 0 no está seguido de un 1)
- 100 (el 0 está al principio pero no está seguido inmediatamente por un 1 al final)

**Árbol de derivación para  $S \rightarrow 1S \mid 01S \mid \epsilon$**

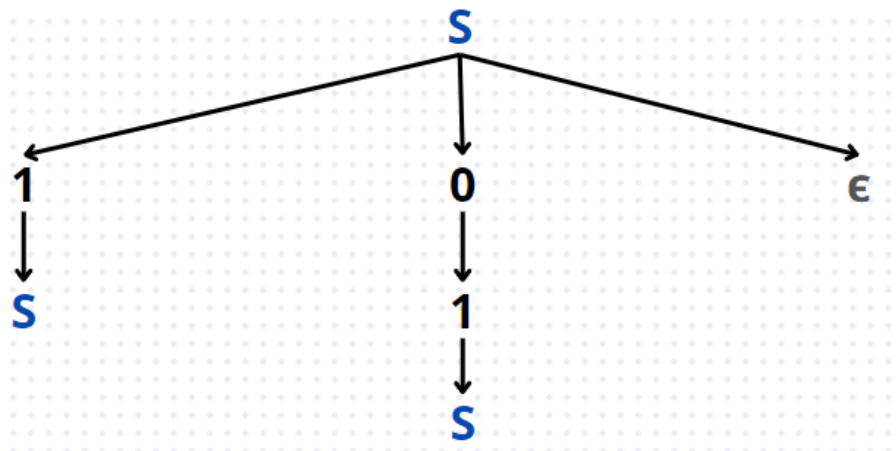


Figure 12: Árbol de la gramática 8

## 10 Ejercicio 10

Elimine la recursividad por la izquierda de la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & (L) \mid a \\ L & \rightarrow & L, S \mid S \end{array} \quad (4)$$

### Eliminar la recursividad en $L$

La regla para  $L$  tiene recursividad por la izquierda en  $L \rightarrow L, S$ , para eliminarla, seguiremos los siguientes pasos:

1. Se escribe  $L$  como  $L'$ , pues  $L'$  será una nueva variable.
2. Separamos las producciones de  $L$  en una parte *recursiva* y *no recursiva*.

**Recursivas:** La variable se llama a sí misma.

**No recursivas:** No se llama a la variable en el lado derecho.

La producción para  $L$  se convierte en:

$$\begin{array}{lcl} S & \rightarrow & (L) \mid a \\ L & \rightarrow & ,SL \mid \epsilon \end{array} \quad (5)$$

### Resultado final

La gramática sin recursividad por la izquierda es:

$$\begin{array}{lcl} S & \rightarrow & (L) \mid a \\ L & \rightarrow & SL \\ L & \rightarrow & ,SL \mid \epsilon \end{array} \quad (6)$$

## 11 Ejercicio 11

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

### Descripción general del pseudocódigo:

El proceso de análisis sintáctico mediante el método descendente recursivo parte de una gramática simple en este caso:

$$S \rightarrow (S) \mid x \quad (7)$$

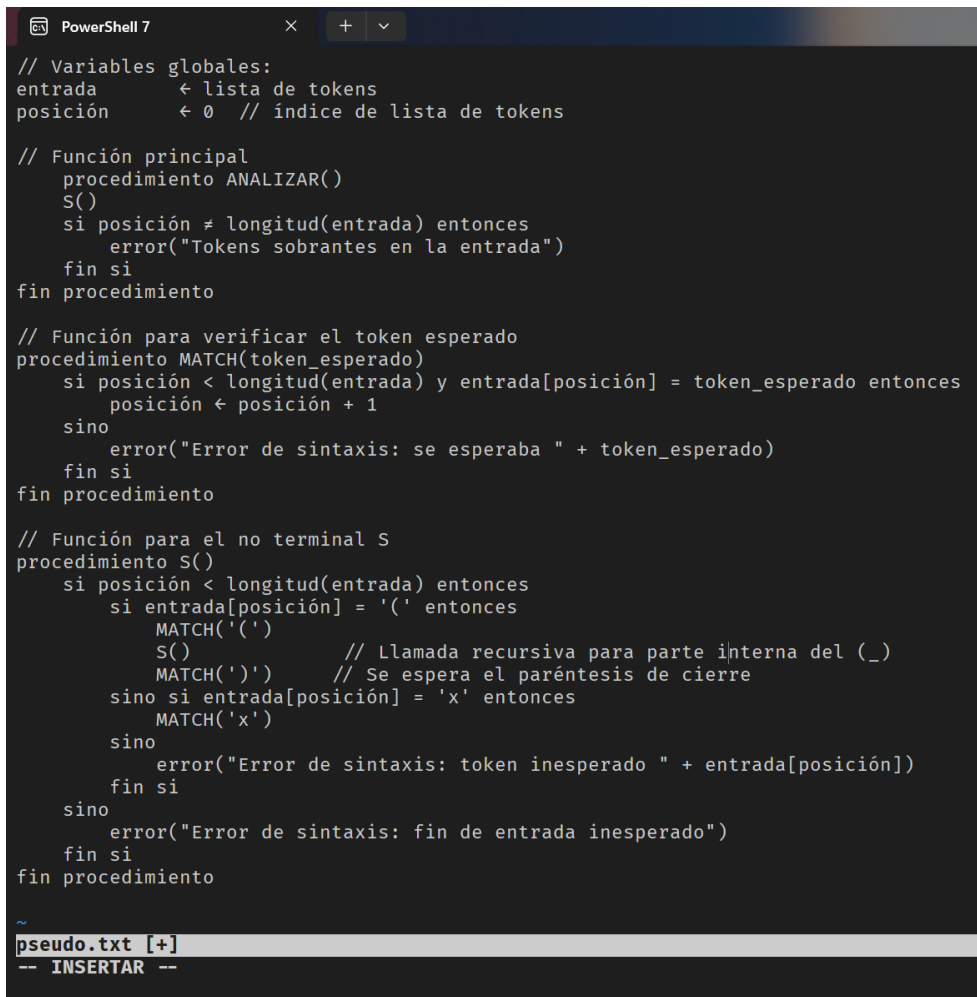
Con este análisis se pretende descender en la estructura de la cadena de entrada según las reglas gramaticales.

El análisis comienza en el símbolo inicial  $S$  y, dependiendo del token actual, decide si la cadena se ajusta a la forma de una expresión entre paréntesis o a la terminal  $x$ .

Esta identificación se lleva a cabo mediante el uso de una función que verifica el token esperado y, en caso de coincidencia, se avanza al siguiente elemento; si no, se reporta un error.

Por lo tanto, la recursividad nos permite analizar estructuras anidadas, ya que la función correspondiente al símbolo  $S$  se invoca nuevamente para evaluar la parte interna de una expresión delimitada por paréntesis. Al finalizar el proceso, se comprueba que toda la entrada haya sido consumida correctamente, lo que garantiza que la cadena analizada cumple con la gramática definida.

### Imagen del pseudocódigo:



```
// Variables globales:
entrada      ← lista de tokens
posición     ← 0 // índice de lista de tokens

// Función principal
procedimiento ANALIZAR()
S()
si posición ≠ longitud(entrada) entonces
    error("Tokens sobrantes en la entrada")
fin si
fin procedimiento

// Función para verificar el token esperado
procedimiento MATCH(token_esperado)
si posición < longitud(entrada) y entrada[posición] = token_esperado entonces
    posición ← posición + 1
sino
    error("Error de sintaxis: se esperaba " + token_esperado)
fin si
fin procedimiento

// Función para el no terminal S
procedimiento S()
si posición < longitud(entrada) entonces
    si entrada[posición] = '(' entonces
        MATCH('(')
        S() // Llamada recursiva para parte interna del ( )
        MATCH(')') // Se espera el paréntesis de cierre
    sino si entrada[posición] = 'x' entonces
        MATCH('x')
    sino
        error("Error de sintaxis: token inesperado " + entrada[posición])
    fin si
sino
    error("Error de sintaxis: fin de entrada inesperado")
fin si
fin procedimiento

~
pseudo.txt [+]
-- INSERTAR --
```

Figure 13: Pseudocódigo

## 12 Ejercicio 12

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)

**Tabla 3.1**

*Tabla de análisis sintáctico para la gramática 3.3*

No terminal	Símbolo de entrada					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Figure 14: Tabla de análisis sintáctico para la gramática 3.3

PILA	ENTRADA	ACCIÓN
\$ E	(id + id) * id \$	$E \rightarrow T E'$
\$ E' T	(id + id) * id \$	$T \rightarrow F T'$
\$ E' T' F	(id + id) * id \$	$F \rightarrow (E)$
\$ E' T' ( E )	(id + id) * id \$	concuerta ( ( )
\$ E' T' E	id + id ) * id \$	$E \rightarrow T E'$
\$ E' T' E' T	id + id ) * id \$	$T \rightarrow F T'$
\$ E' T' E' T' F	id + id ) * id \$	$F \rightarrow id$
\$ E' T' E' T' id	id + id ) * id \$	concuerta ( id )
\$ E' T' E' T'	+ id ) * id \$	$T' \rightarrow \varepsilon$
\$ E' T' E'	+ id ) * id \$	concuerta ( + )
\$ E' T' T	id ) * id \$	$T \rightarrow F T'$
\$ E' T' F	id ) * id \$	$F \rightarrow id$
\$ E' T' id	id ) * id \$	concuerta ( id )
\$ E' T'	) * id \$	$T' \rightarrow \varepsilon$
\$ E'	) * id \$	$E' \rightarrow \varepsilon$
\$ E	) * id \$	concuerta ( ) )
\$ T	* id \$	$E \rightarrow T E'$
\$ T'	* id \$	concuerta ( * )
\$ F	id \$	$T' \rightarrow F T'$
\$ id	id \$	$F \rightarrow id$
\$ F	\$	concuerta ( id )
\$ T'	\$	$T' \rightarrow \varepsilon$
\$ E'	\$	$E' \rightarrow \varepsilon$
\$ E	\$	<b>Aceptar</b>

Table 1: Tabla de análisis descendente

## 13 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

$$F \rightarrow (E) \mid id \mid num \quad (8)$$

La gramática inicial solo maneja las operaciones de suma y multiplicación:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

### Modificación para Incluir Resta y División:

Para agregar las operaciones de resta y división, extendemos las reglas de producción:

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

### Eliminación de Recursión por la Izquierda:

Identificamos que existe recursión por la izquierda en las producciones de  $E$  y  $T$ , por lo que las reescribimos utilizando variables auxiliares  $E'$  y  $T'$ :

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid -TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid /FT' \mid \epsilon \\ F &\rightarrow (E) \mid id \mid num \end{aligned}$$

### Gramática Final:

La gramática resultante, sin recursión por la izquierda y con las nuevas operaciones, es:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid -TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid /FT' \mid \epsilon \\ F &\rightarrow (E) \mid id \mid num \end{aligned}$$

Esta gramática ahora admite suma, resta, multiplicación y división, sin recursión por la izquierda, y permite que  $F$  derive en números.

## 14 Ejercicio 14

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

**Pseudocodigo:**

```
// Analizador descendente recursivo para la gramática

procedure E()
    T()
    E'()

procedure E'()
    if token_actual = '+' or token_actual = '-' //suma y resta
        consumir(token_actual)
        T()
        E'()
    else
        return // caso ε

procedure T()
    F()
    T'()

procedure T'()
    if token_actual = '*' or token_actual = '/' // multi y divi
        consumir(token_actual)
        F()
        T'()
    else
        return // ε (epsilon)

procedure F() //para paréntesis
    if token_actual = '('
        consumir('(')
        E()
        consumir(')')
    else if token_actual = 'id' or token_actual = 'num'
        consumir(token_actual)
    else
        error("Token inesperado")

~
~
~
~
pseudo.txt [+]
-- INSERTAR --
```

Figure 15: Pseudocódigo Método Descendente Recursivo

### Código de la implementación en Rust

```
use std::io::{self, Write};

//estructura con lista de tokens
#[derive(Debug)]
struct Parser{
    tokens:Vec<String>, //lista
    index: usize, //apuntador int din mico
}

//m todos del parser
impl Parser{
    //inicializa parser con su lista e indice
    fn new(tokens: Vec<String>)->Self{
        Parser{tokens, index:0}
    }
}
```

```

//toma el token con el index actual
fn token_actual(&self) -> Option<&String>{
    self.tokens.get(self.index)
}

//avanza del token actual si se cumple la condici n
fn consumir(&mut self, esperado: &str) -> Result<(), String> {
    if let Some(token) = self.token_actual(){
        if token == esperado {
            self.index += 1;
            Ok(())
        }else{
            Err(format!(
                "Error de sintaxis: Se esperaba '{}', pero se encontr { }'",
                esperado, token))
        }
    }else {
        Err("Error de sintaxis: Fin inesperado de entrada.".to_string())
    }
}

//Parseo
//Comienza analisis con e()
fn parse(&mut self) -> Result<(), String> {
    self.e()?; // Llamada a 'e'
    if self.index < self.tokens.len() {
        return Err("Error de sintaxis: Tokens extra no procesados.".to_string());
    }
    println!("An lisis sint ctico exitoso.");
    Ok(())
}

fn e(&mut self) -> Result<(), String> {
    self.t()?; // Llamada a T
    self.e_()?; // Llamada a E_
    Ok(())
}

fn e_(&mut self) -> Result<(), String> {
    if let Some(token) = self.token_actual(){
        let token_str = token.clone();

        if token_str == "+" || token_str == "-" {
            self.consumir(&token_str)?; // Consumir token
            self.t()?; // Llamada a T
            self.e_()?; // Llamada recursiva a E_
        }
    } // caso - no hacemos nada
    Ok(())
}

fn t(&mut self) -> Result<(), String> {
    self.f()?; // Llamada a F
    self.t_()?; // Llamada a T_
    Ok(())
}

fn t_(&mut self) -> Result<(), String> {
    if let Some(token) = self.token_actual() {
        let token_str = token.clone(); // clonar token
        if token_str == "*" || token_str == "/" {
            self.consumir(&token_str)?; // Consumiendo el token
            self.f()?; // Llamada a F
            self.t_()?; // Llamada recursiva a T_
        }
    }
    // , no hacemos nada
    Ok(())
}

fn f(&mut self) -> Result<(), String> {
    if let Some(token) = self.token_actual() {
        let token_str = token.clone();
        if token_str == "(" {
            self.consumir(&token_str)?; // Consumiendo '('
            self.e()?; // Llamada a 'E' para procesar dentro de par ntesis
        }
    }
}

```

```

        self.consumir(")?; // Consumiendo ')'
    } else if token_str.chars().all(char::is_alphanumeric) {
        self.consumir(&token_str)?; // Consumiendo identificador o número
    } else {
        return Err(format!("Error de sintaxis: Token inesperado '{}'", token));
    }
} else {
    return Err("Error de sintaxis: Fin inesperado de entrada.".to_string());
}
Ok(())
}
}

// Tokenizar entrada
fn tokenizar(input: &str) -> Vec<String> {
    input.split_whitespace().map(|s| s.to_string()).collect()
}

fn main() {
    // Leer la entrada del usuario
    print!("Introduce una expresión matemática: ");
    io::stdout().flush().unwrap(); // Mensaje impreso antes de la lectura

    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Error al leer la línea");
    let input = input.trim(); // Eliminar saltos de línea

    // Tokenizar la entrada
    let tokens = tokenizar(input);

    let mut parser = Parser::new(tokens);
    if let Err(err) = parser.parse() {
        eprintln!("{}", err);
    }
}

```

**Link Github:** <https://github.com/slenderxd/Aut-matas-y-Compiladores/blob/main/Léxico/Ejercicio14.rs>