LICIENCIATURA EN CIENCIAS COMPUTACIONALES
AUTÓMATAS Y COOMPILADORES

REPORTE DE PRÁCTICA 2.4

ANÁLISIS SINTÁCTICO - EJERICICIOS

ALUMNO: CHRISTIAN LÓPEZ SOLÍS

DR. EDUARDO CORNEJO VELÁZQUEZ

26 DE MARZO DEL 2025

MINERAL DE LA REFORMA, PACHUCA

1. Introducción

En la materia de Autómatas y Compiladores, el análisis sintáctico es un proceso fundamental para la construcción de compiladores y el reconocimiento de la estructura gramatical de un lenguaje de programación. Este análisis se encarga de verificar que una secuencia de tokens generada en la fase léxica cumpla con las reglas sintácticas del lenguaje, garantizando así su correcta organización jerárquica. Existen distintos enfoques para el análisis sintáctico, como el análisis descendente, que construye el árbol sintáctico desde la raíz hacia las hojas, y el análisis ascendente, que parte de los elementos básicos para formar estructuras más complejas. Ambos métodos son esenciales para interpretar el código fuente de manera precisa y eficiente.

2. Objetivo

El objetivo principal es comprender el proceso de análisis sintáctico y su importancia en la construcción de compiladores. Se busca identificar los diferentes métodos de análisis sintáctico, como el descendente y el ascendente, así como aprender a construir árboles sintácticos para representar la estructura gramatical de un lenguaje. Además, se pretende aplicar este conocimiento en el diseño e implementación de analizadores sintácticos que validen la correcta organización de los elementos en el código fuente.

3. Marco Teórico

Análisis Sintáctico

El análisis sintáctico es una fase del proceso de compilación que se encarga de verificar que una secuencia de tokens generada en la fase léxica cumpla con las reglas gramaticales del lenguaje de programación. Su función principal es construir una representación jerárquica de la estructura del programa, conocida como árbol sintáctico.

Gramáticas Libres de Contexto (GLC)

Las gramáticas libres de contexto son un tipo de gramática formal que se utiliza ampliamente en el análisis sintáctico. Se definen mediante un conjunto de reglas de producción que establecen cómo se pueden derivar las cadenas válidas del lenguaje. Cada regla está compuesta por un símbolo no terminal que puede ser sustituido por una secuencia de símbolos terminales y/o no terminales.

Métodos de Análisis Sintáctico

Existen dos enfoques principales en el análisis sintáctico:

- **Análisis Descendente:** Este método construye el árbol sintáctico desde la raíz hacia las hojas. Se basa en técnicas como el análisis predictivo y el análisis LL(k), que requieren gramáticas bien definidas para evitar ambigüedades.
- **Análisis Ascendente:** En este enfoque, el árbol sintáctico se construye desde las hojas hacia la raíz. El análisis LR(k) es una técnica comúnmente utilizada que permite manejar una amplia variedad de gramáticas libres de contexto.

Árboles Sintácticos

Un árbol sintáctico es una representación gráfica que organiza jerárquicamente los elementos del programa según las reglas gramaticales del lenguaje. Cada nodo interno representa un símbolo no terminal, mientras que las hojas representan símbolos terminales. Esta estructura facilita la detección de errores y la posterior generación de código intermedio.

4. Herramientas Empleadas

Libro

Se utilizaron recursos educativos como un Libro en la materia para comprender y profundizar en temas como es el análizador sintáctico. Proporcionando una forma accesible y visual de explicar estos conceptos complejos, junto con la defínción de ejemplos prácticos.

Editor de Textos LaTeX

Para la redacción del informe y la organización del marco teórico, se empleó LaTeX, una herramienta eficiente para la creación de documentos científicos, que permite organizar y estructurar contenido de manera clara y precisa, con un manejo adecuado de la información.

Generador de árboles sintácticos RSyntaxTree

Para la generación de árboles sintácticos, se empleó RSyntaxTree, una herramienta eficiente para la creación de estos módelos, que permite organizar y estructurar contenido de manera clara y precisa, con un manejo del análisis.

Referencia:

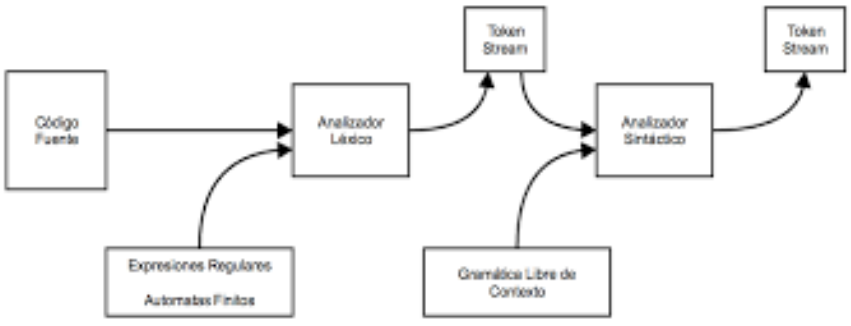

Figure 1: Análizador Sintáctico.

5. Desarrollo

Ejercicio 1:

Planteamiento a: Escriba una gramática que genere el conjunto de cadenas.

Conjunto: { s; , s;s; , s;s;s; , . . . }
G = (N, Σ, P, S)

- **Donde:**
  - **Conjunto de no terminales (N):** { S }
  - **Conjunto de terminales (sum):** { s , ; }
  - **Conjunto de reglas de producción (P):** { S → s; , S → S s; }
  - **Símbolo inicial (S):** { S }

- **Gramática generada:** S → s; | S → Ss;

Planteamiento b: Genere un árbol sintáctico para la cadena s;s; .

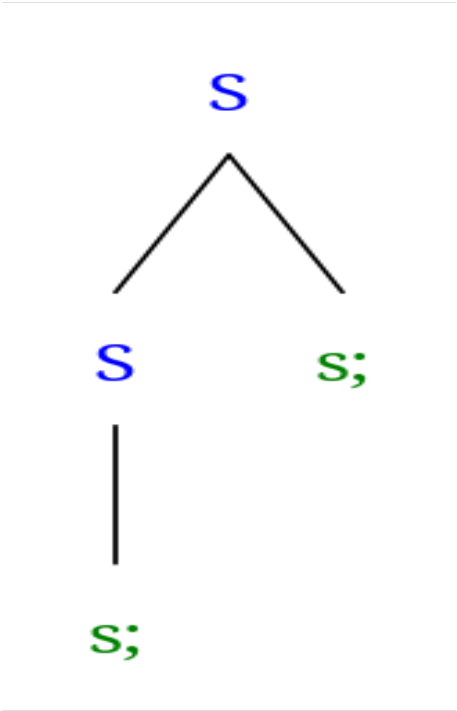

Figure 2: Árbol Sintáctico.

Ejercicio 2:

Planteamiento a: Considere la siguiente gramática:

rexp → rexp “|” rexp

- → rexp rexp
- → rexp ”*”
- → ”(” rexp ”)”
- → letra

Árbol Sintáctico generado para la expresión regular (ab l b) * :

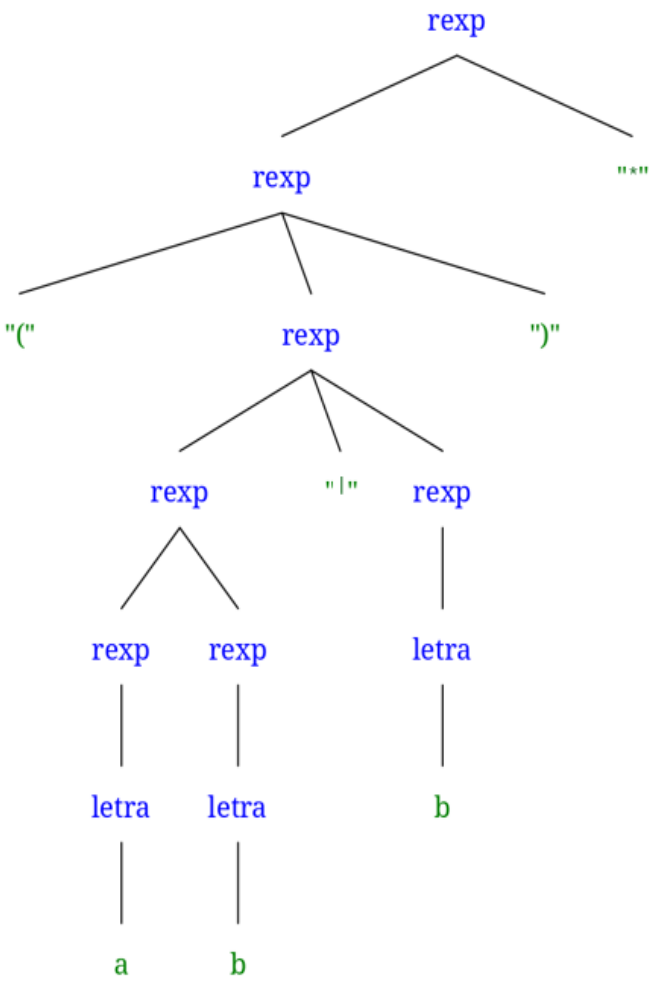

Figure 3: Árbol Sintáctico.

Ejercicio 3:

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

a) S → S S + | S S * | a con la cadena aa+a*.

- **Descripción del lenguaje generado:**
  - **a)** Esta gramática genera expresiones algebraicas compujestas por la letra a y los operadores + y *.
  - **b)** Cada S puede ser reemplazada por una a o por una combinación de S con operadoes + o *, lo que permite formar expresiones en notación postfija.
  - **c)** Se 'pueden encandenar múltiles a con + y * siguiendo las reglas de derivación.

- **Lenguaje generado:** { aa+a* }

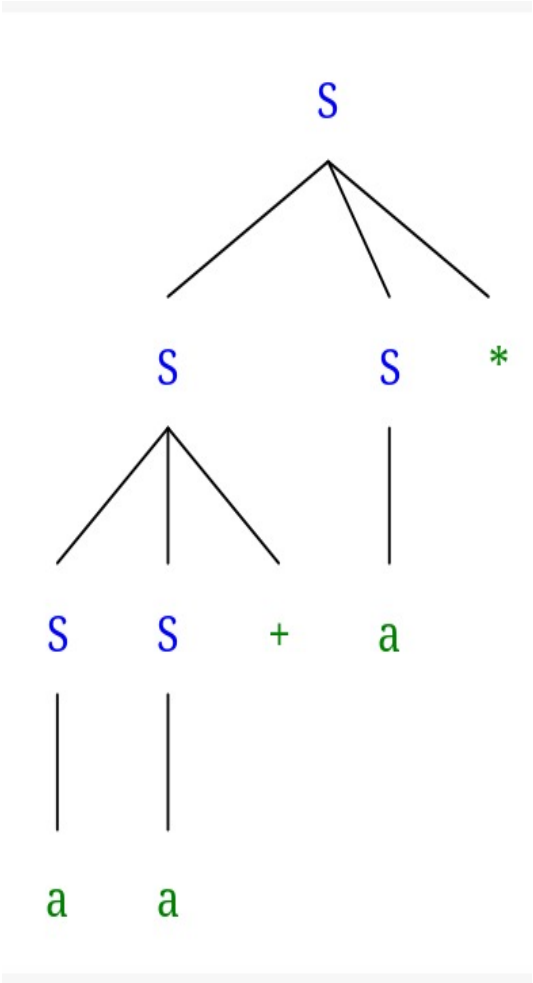

Figure 4: Árbol Sintáctico.

b) S → 0 S 1 | 0 1 con la cadena 000111.

- **Descripción del lenguaje generado:**
  - **a)** Esta gramática genera cadenas que contienen el mismo número de ceros (0) y unos (1), con cada 0 emparejando con un 1.
  - **b)** Además de que la recursión en S → 0S1 permite formar cadenas más largas con la estructura anidada.
  - **c)** Cadenas balanceadas en las que cada 0 tiene un 1, correspondiente al final.

- **Lenguaje generado:** { 000111 }

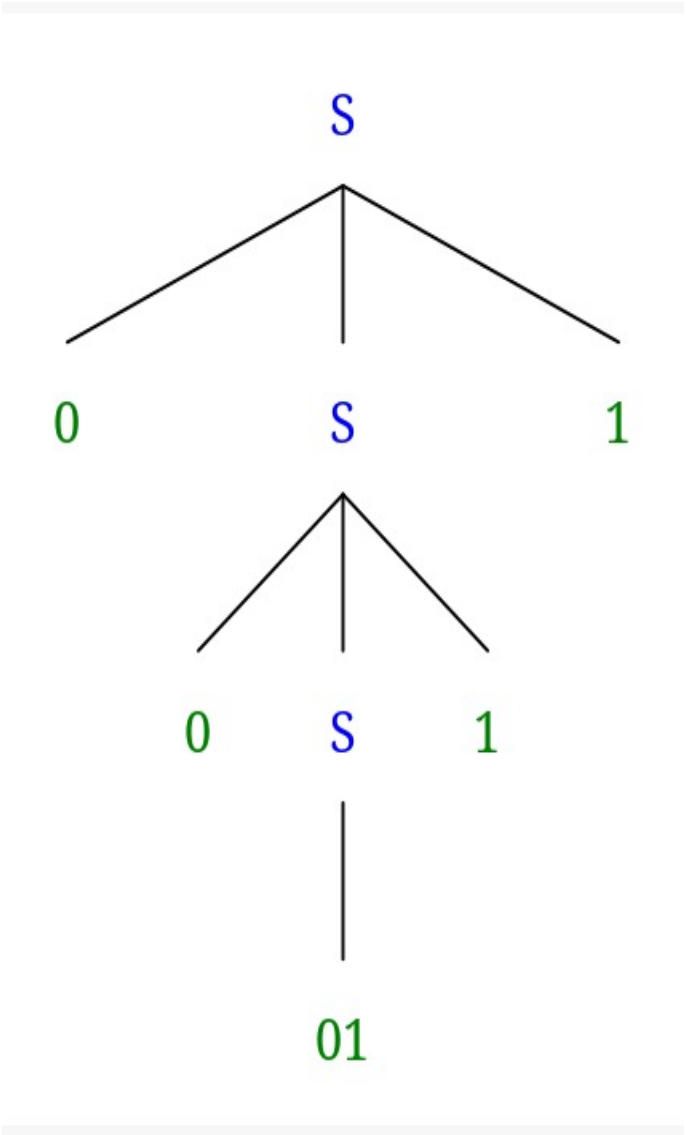

Figure 5: Árbol Sintáctico.

c) S → + S S | * S S | a con la cadena + * aaa

- **Descripción del lenguaje generado:**
  - **a)** Esta gramática genera expresiones en notación prefija (notación polaca), donde cada operador + o * aparece antes de los operandos.
  - **b)** Un S puede ser una a o una expresión que comience con + o * seguido de dos S.
  - **c)** Este lenguaje produce expresiones en notación prefija (polaca), en las que los operadores (+, *) preceden a los operandos o subexpresiones (a).

- **Lenguaje generado:** { + * aaa }

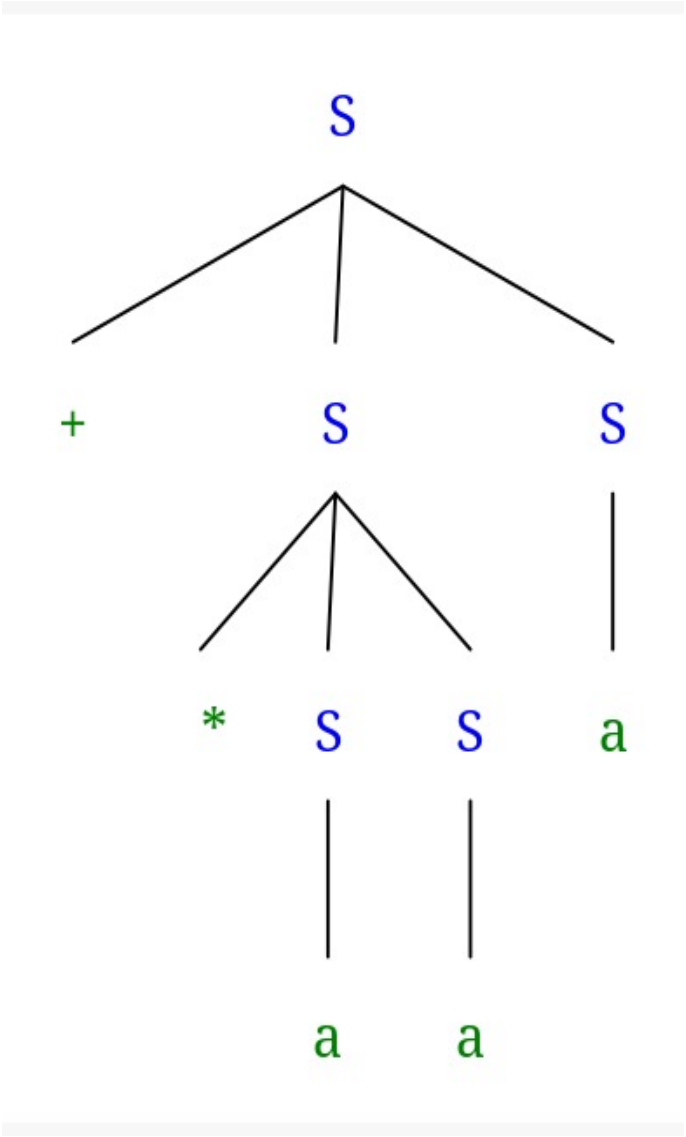

Figure 6: Árbol Sintáctico.

Ejercicio 4:

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

Grámatica: $S \rightarrow xSy \mid \epsilon$

- **Descripción:**
  - **S → xSy:** Esta regla genera una cadena que comienza con x, seguida de una subcadena generada por S, y luego termina con y. Esta regla permite que el número de x's sea igual al número de y's, con las x's siempre precediendo a las y's.
  - **S → ϵ:** Esta regla genera la cadena vacía en el ultimo nivrl aplicado, además de no que no representa un espacio en blaco, simplemente se elimina como si no existiera..
- **Conclusión:** $x^n y^n para n 0; donde nes el número de x y emparejados.$

- **Lenguaje generado:** { E, xy, xxyy, xxxyyy, ... }

Ejemplo de árbol:

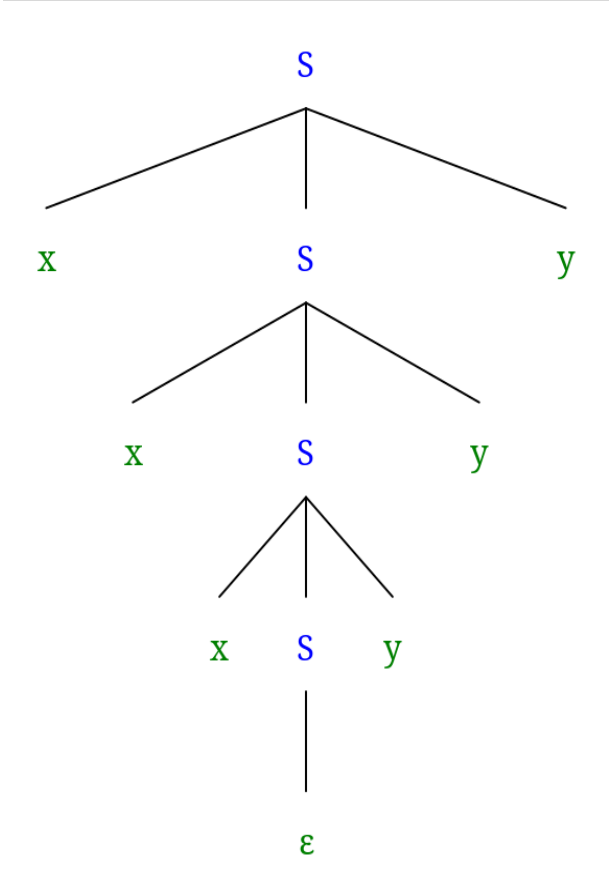

Figure 7: Árbol Sintáctico.

Ejercicio 5:

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

Grámatica 1: S → zMNz
Grámatica 2: M → aNa
Grámatica 3: N → bNb
Grámatica 4: N → z

Árbol Sintáctico generado para la cadena zazabzbz:

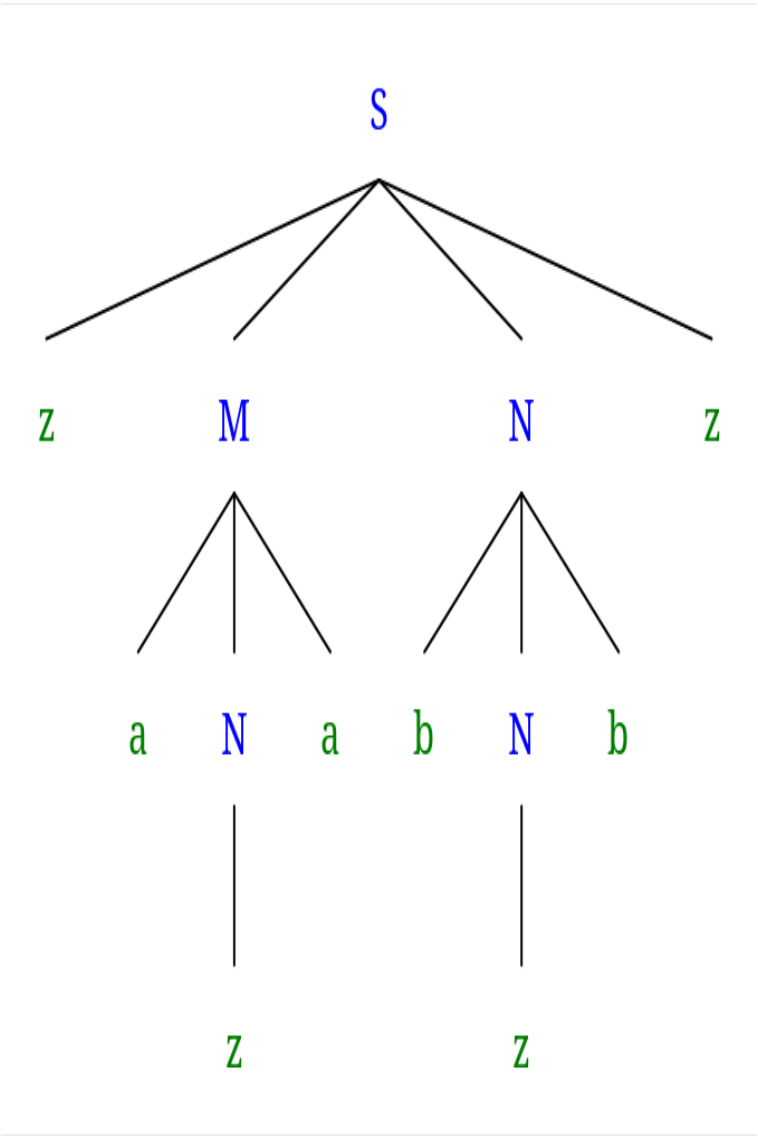

Figure 8: Árbol Sintáctico.

Ejercicio 6:

Planteamiento : 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:

Grámatica 1: S → ictS
Grámatica 2: S → ictSeS
Grámatica 3: S → s

Solución: Para demostrar que la gramática es ambigua, necesitamos mostrar que la cadena ictictses tiene al menos dos derivaciones que producen diferentes árboles de análisis sintáctico. Por lo que:

- **Derivación 1 de la cadena utilizando la regla S → ictS:**
  - **La raíz es S, que se descompone en i,c,t y otro S:** Usamos la regla: S → ictS
  - **La S del nivel 1 se descompone en i,c,t,S,e, S:** Usamos la regla: S → ict(ictSeS)
  - **El primer S interno dentro de este segundo nivel deriva en s:** Usamos la regla: S → ict(ict(s)eS)
  - **El segundo S interno también deriva en s:** Usamos la regla: S → ict(ict(s)e(s))
- **Siguiendo el árbol, la cadena generada es:**
  - **Cadena derivada:** ictictses

Árbol Sintáctico generado para la cadena zazabzbz:

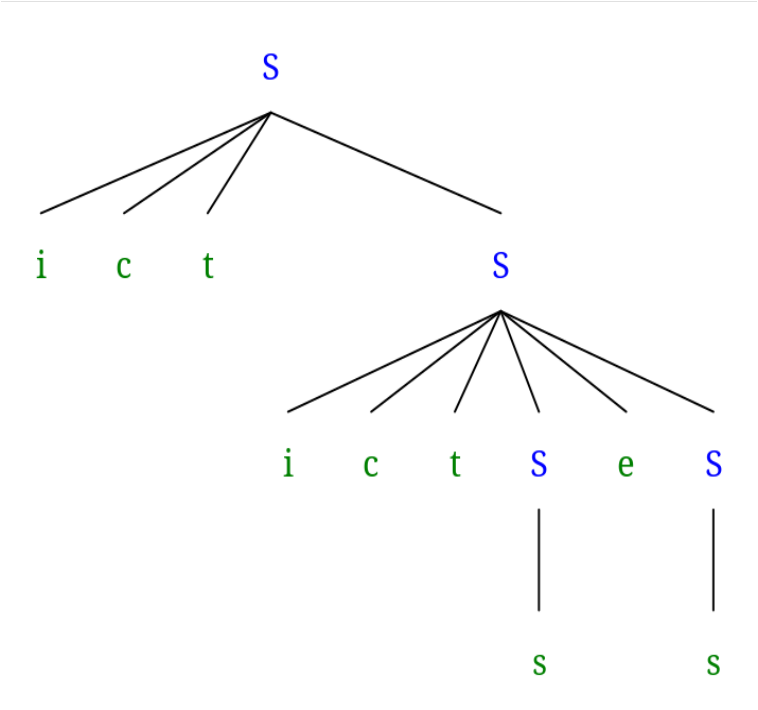

Figure 9: Árbol Sintáctico.

- **Derivación 2 de la cadena utilizando la regla S → ictSeS:**
  - **La raíz es S, que se descompone en i,c,t, S, e y otro S:** Usamos la regla: S → ictSeS
  - **El primer S interno dentro de este primer nivel deriva en i, c, t, y otro S:** Usamos la regla: S → ict(ictS)eS)
  - **El segundo S interno de este primer nivel también deriva en s:** Usamos la regla: S → ict(ict(ictS)e(s))
  - **Ahora la siguinte S interna de nuestro segundo nivel deriva en s:** Usamos la regla: S → ict(ict(ict(s))e(s))
- **Siguiendo el árbol, la cadena generada es:**
  - **Cadena derivada:** ictictses

Árbol Sintáctico generado para la cadena zazabzbz:

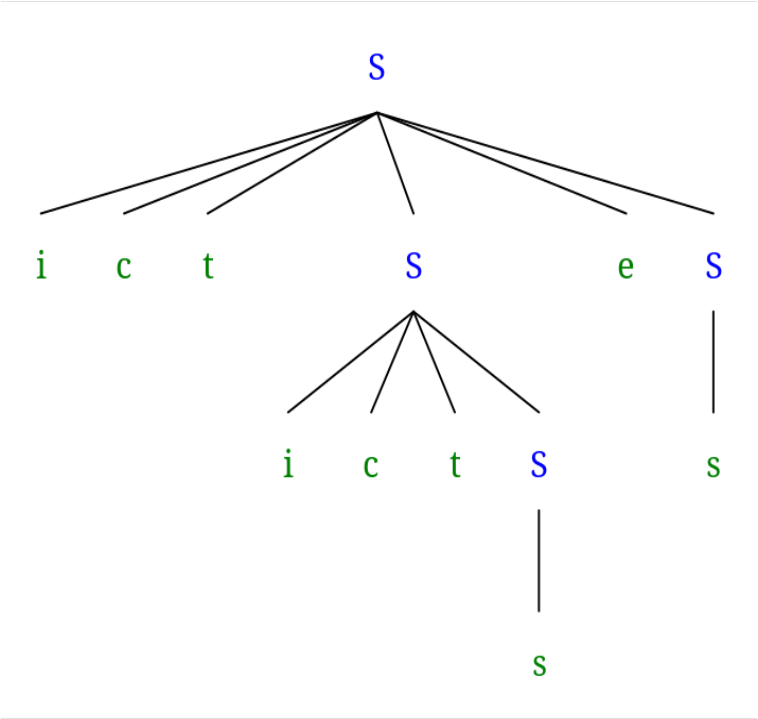

Figure 10: Árbol Sintáctico.

Conclusión: Hemos demostrado que la cadena ictictses tiene al menos dos derivaciones o más estructuras sintácticas distintas. En este caso, para la comprobación se observa claramente en los dos árboles de derivación que representan formas diferentes de agrupar los símbolos, aunque pueden existir más árboles, estos son dos de muestra.

Ejercicio 7:

Planteamiento: Considere la siguiente gramática:

Grámatica 1: S → (L) | a
Grámatica 2: L → L , S | S

- **Encuéntrense árboles de análisis sintáctico para las siguientes frases:**
  - **a)** (a, a)

- **Árbol generado:**

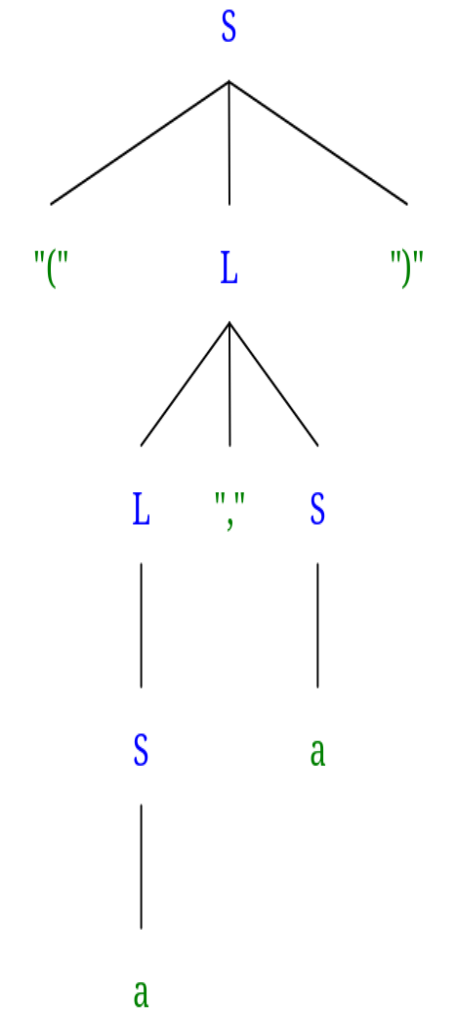

Figure 11: Árbol Sintáctico.

- **Encuéntrense árboles de análisis sintáctico para las siguientes frases:**
  - **b)** (a, (a, a))

- **Árbol generado:**

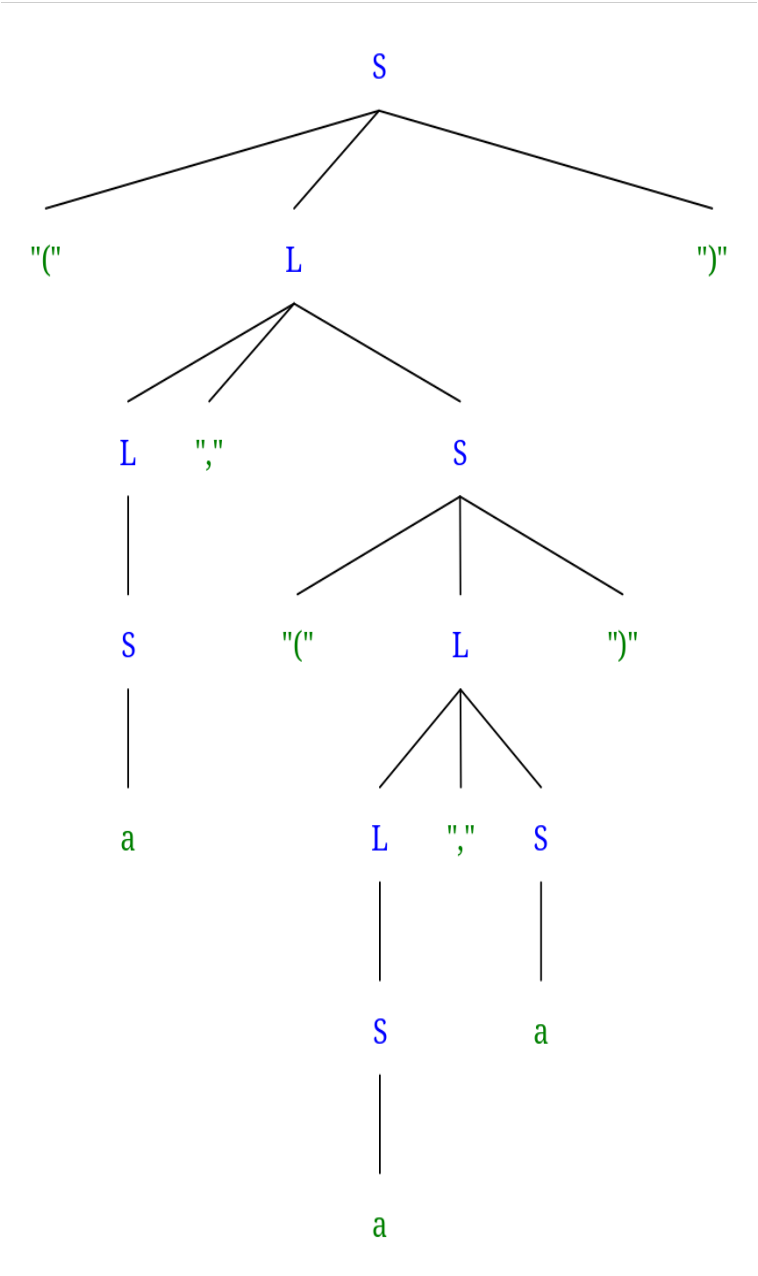

Figure 12: Árbol Sintáctico.

- **Encuéntrense árboles de análisis sintáctico para las siguientes frases:**
  - **c)** (a, ((a, a), (a, a)))

- **Árbol generado:**

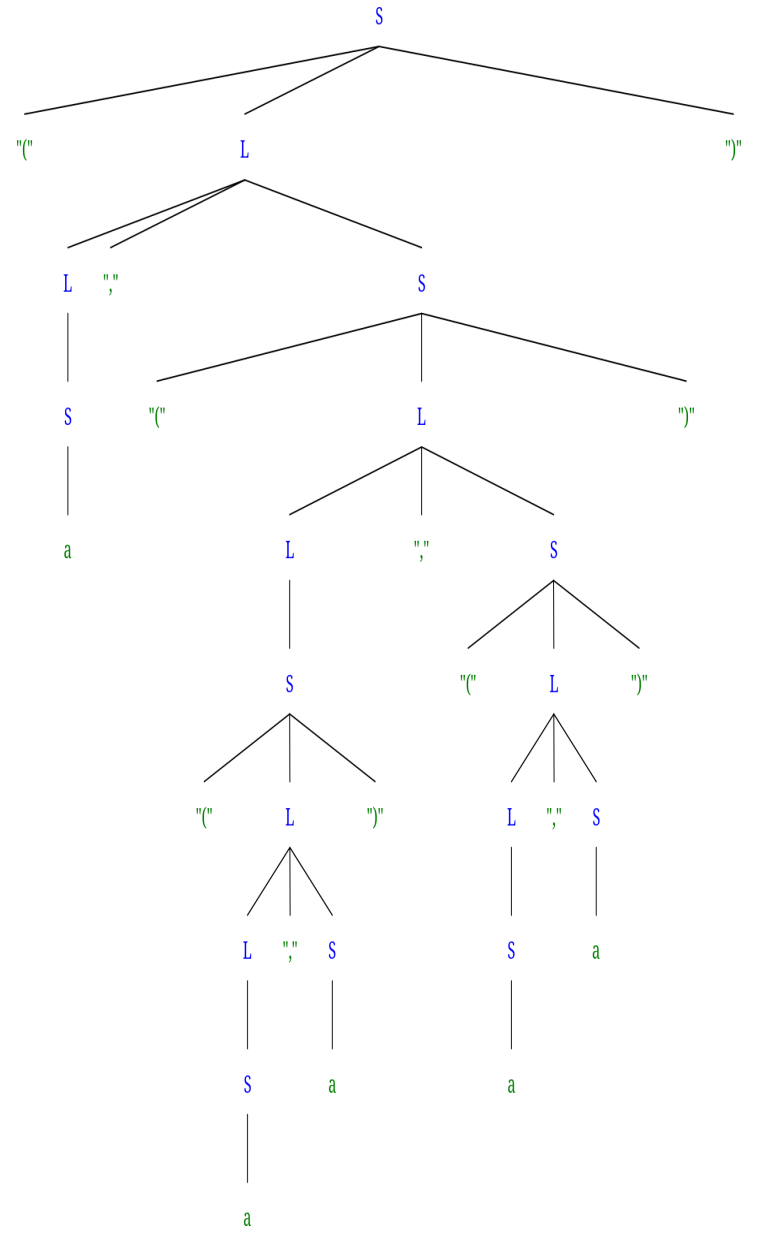

Figure 13: Árbol Sintáctico.

Ejercicio 8:

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

bexpr → bexpr or bterm | bterm

- **bterm** → bterm and bfactor | bfactor
- **bfactor** → not bfactor | (bexpr) | true | false

Árbol Sintáctico generado para la frase not (true or false):

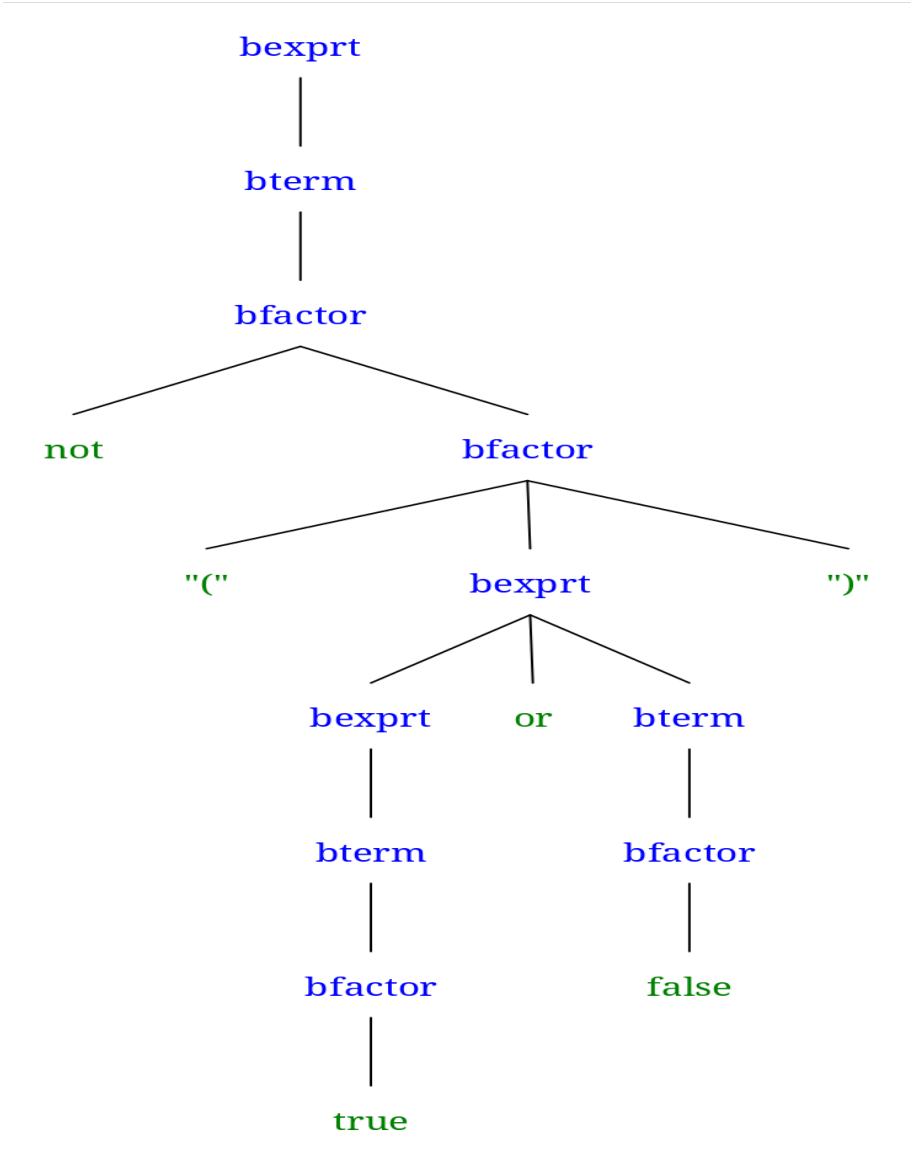

Figure 14: Árbol Sintáctico.

Ejercicio 9:

Planteamiento a: 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:

Conjunto: { 1, 01, 011, 0111101 ... }
G = (N, Σ, P, S)

- **Donde:**
  - **Conjunto de no terminales (N):** { S }
  - **Conjunto de terminales (sum):** { 0, 1 }
  - **Conjunto de reglas de producción (P):** { S $\rightarrow$ 1S; , S $\rightarrow$ 01S; , S $\rightarrow \epsilon$; }
  - **Símbolo inicial (S):** { S }

- **Gramática generada:**
  - **S $\rightarrow$ 1S;**
  - **S $\rightarrow$ 01S;**
  - **S $\rightarrow \epsilon$;**

Ejemplo de árbol:

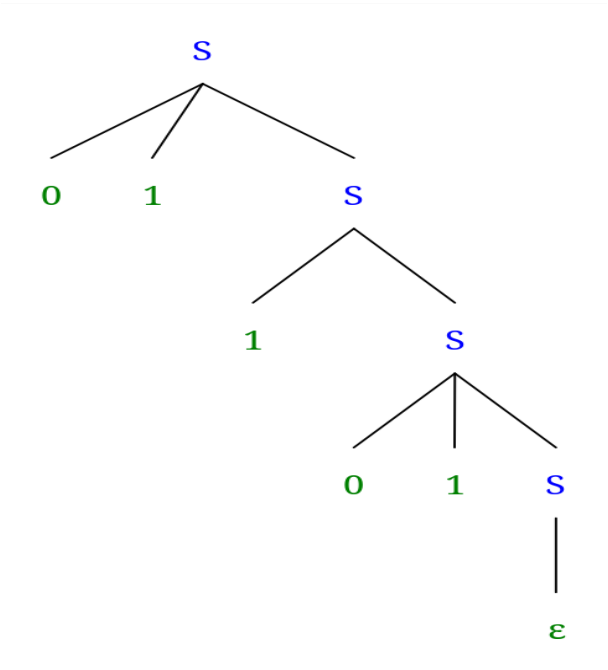

Figure 15: Árbol Sintáctico.

Ejercicio 10:

Planteamiento: Elimine la recursividad por la izquierda de la siguiente gramática:

Gramática original: bexpr or bterm l bterm

- **S** → (L) | a
- **S** → L , S | S

Eliminación de la recursividad por la izquierda:
a) La primer grámtica esta correcta, no hay recursividad directa por la izquierda en S, ya que ninguna producción de S comienza con S.
b) La segunda grámtica, tiene recursividad por la izquierda porque comienza con L, lo que puede llevar a una derivación infinita en algunos analizadores sintácticos.

Eliminación de la recursividad para la gramática: S → L , S | S:

- **Dividmos las reglas:**
  - **L**a parte que contiene la recursividad L → L , S .
  - **L**a parte sin recursividad L → S .

- **Reescribimos L introducinedo un nuevo no terminal, en este caso L' para manejar la recursividad:**
  - **L** → S L'
  - **L'** → , S L' | ϵ

- **Nueva gramática:**
  - **S** → (L) | a
  - **L** → S L'
  - **L'** → , S L' | ϵ

- **Explicación:**
  - **A**hora, L genera primero S seguido de L'.
  - **L**' maneja las partes recursivas ,S y permite terminar con E, evitando la recursividad directa por la izquierda.

Ejercicio 11:

Planteamiento: Dada la gramática S → (S) | x, escriba un pseudocódigo para el análisis sintáctico de esta gramática mediante el método descendente recursivo:

Pseudocódigo:

- **Función recursivaS():**
  - **S**i la entrada comienza con '(':
    - **C**omienza una nueva llamada recursiva a recursivaS() para analizar el contenido dentro de los paréntesis.
    - **S**i la entrada no termina con ')', el análisis falla.
    - **S**i después de ')', no hay más símbolos, el análisis es exitoso.
  - Sino, si la entrada comienza con 'x':
    - **C**onsume el símbolo 'x' y termina el análisis exitosamente.
  - **S**ino:
    - **E**l análisis falla (la entrada no coincide con ninguna de las producciones de S).
- **Función recursiva(entrada):**
  - **L**lamar a recursivaS() con la cadena de entrada.
  - **S**i recursivaS() termina con éxito y no quedan símbolos no analizados, el análisis es exitoso.
  - **S**ino, el análisis falla.

Ejercicio 12:

Planteamiento : 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):

Algoritmo 3.2 para el análisis sintáctico de la gramática

- **E**l algoritmo 3.2 permite llevar acabo el análisi sintáctico predictivo en el contexto de la compilación y el procesamiento de lenguajes formales. Su propósito es verificar si una cadena de entrada pertenece al lenguaje definido por una gramática utilizando una tabla de análisis sintáctico.

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 ($X \neq \$$) { /* 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;
}

Table 1: Algoritmo de análisis sintáctico para la gramática.

Tabla de análisis sintáctico

- **E**sta tabla permite que el Algoritmo 3.2 se ejecute de forma eficiente y sin ambigüedades. Su diseño garantiza que el analizador predictivo seleccione siempre la producción correcta en función del símbolo de entrada, evitando decisiones erróneas o bucles infinitos.

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 2: Tabla de análisis sintáctico para la gramática.

Solución a la grámatica porpuesta:

- **Moviminetos que realiza un analizador sintáctico predicitivo:**
  - **Entrada:** (id + id) * id

Pila	Entrada	Acción
$ E	(id + id) * id $	E → T E’
$ E’ T	(id + id) * id $	T → F T’
$ E’ T’ F	(id + id) * id $	F → (E)
$ E’ T’) E (	(id + id) * id $	concuerda (“(”)
$ E’ T’) E	id + id) * id $	E → T E′
$ E’ T’) E’ T	id + id) * id $	T → F T′
$ E’ T’) E’ T’ F	id + id) * id $	F → id
$ E’ T’) E’ T’ id	id + id) * id $	concuerda (“id”)
$ E’ T’) E’ T’	+ id) * id $	T′ → ε
$ E’ T’) E’	+ id) * id $	E’ → + T E′
$ E’ T’) E’ T +	+ id) * id $	concuerda (“+”)
$ E’ T’) E’ T	id) * id $	T → F T′
$ E’ T’) E’ T’ F	id) * id $	F → id
$ E’ T’) E’ T’ id	id) * id $	concuerda (“id”)
$ E’ T’) E’ T’	) * id $	T′ → ε
$ E’ T’) E’	) * id $	E′ → ε
$ E’ T’)	) * id $	concuerda (“)”)
$ E’ T’	* id $	T’ → * F T′
$ E’ T’ F *	* id $	concuerda “*”
$ E’ T’ F	id $	F → id
$ E’ T’ id	id $	concuerda (“id”)
$ E’ T’	$	T′ → ε
$ E’	$	E′ → ε
$	$	Aceptar ()

Table 3: Moviminetos de un analizador sintáctico predicitivo.

Ejercicio 13:

Planteamiento: 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 original:

- **E** → E + T | T
- **T** → T * F | F
- **F** → (E) | id

Gramática modificada:

- **E** → E + T | E - T | T
- **T** → T * F | T / F | F
- **F** → (E) | id | num

Eliminación de la recursividad por la izquierda:
a) La primera y segunda grámtica, tiene recursividad por la izquierda porque comienza con E y T, lo que puede llevar a una derivación infinita en algunos analizadores sintácticos.
b) La tercera grámtica esta correcta, no hay recursividad directa por la izquierda en S, ya que ninguna producción de S comienza con S.

Eliminación de la recursividad para la primera y segunda gramática:

- **Dividimos las reglas:**
  - **L**a parte que contiene la recursividad E → E + T | E - T | T y T → T * F | T / F | F .
  - **L**a parte sin recursividad F → (E) | id | num .

- **Reescribimos E y T introducinedo un nuevo no terminal, en este caso E' y T' para manejar la recursividad:**
  - **E** → T E'
  - **E'** → + T E' | - T E' | ϵ
  - **T** → F T'
  - **T'** → * F T' | / F T' | ϵ

- **Nueva gramática:**
  - **E** → T E'
  - **E'** → + T E' | - T E' | ϵ
  - **T** → F T'
  - **T'** → * F T' | / F T' | ϵ
  - **F** → (E) | id | num

Ejercicio 14:

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

Pseudocódigo:

```
Inicio
  Leer expresión desde el usuario
  Intentar
    Crear objeto Parser con la expresión
    Llamar al método analizar del objeto Parser
    Imprimir "Expresión válida."
  Capturar excepción RuntimeException como e
    Imprimir el mensaje de error de la excepción
Fin

Clase Parser
  Atributos:
    input: cadena de texto que contiene la expresión
    index: índice actual en la expresión
    tokenActual: el token actual de la expresión

  Constructor (input)
    Eliminar espacios en blanco de la expresión
    Establecer tokenActual como el siguiente token de la expresión

  Método obtenerSiguienteToken
    Si index es menor que la longitud de la expresión
      Devolver el carácter en la posición index
    Sino
      Devolver el carácter nulo ('\0')

  Método analizar
    Llamar al método E
```

```
  Si tokenActual no es '\0'
    Llamar al método error con el mensaje "Expresión mal formada"

Método E
  Llamar al método T
  Llamar al método E_

Método E_
  Si tokenActual es '+' o '-'
    Obtener el operador actual
    Establecer tokenActual como el siguiente token
    Llamar al método T
    Llamar al método E_

Método T
  Llamar al método F
  Llamar al método T_

Método T_
  Si tokenActual es '*' o '/'
    Obtener el operador actual
    Establecer tokenActual como el siguiente token
    Llamar al método F
    Llamar al método T_

Método F
  Si tokenActual es '('
    Establecer tokenActual como el siguiente token
```

```
        Llamar al método E
        Si tokenActual es ')'
            Establecer tokenActual como el siguiente token
        Sino
            Llamar al método error con el mensaje "Se esperaba ')'."
    Sino, Si tokenActual es un dígito
        Mientras tokenActual sea un dígito
            Establecer tokenActual como el siguiente token
    Sino, Si tokenActual es una letra
        Establecer tokenActual como el siguiente token
    Sino
        Llamar al método error con el mensaje "Token inesperado."

  Método error(mensaje)
    Lanzar una excepción RuntimeException con el mensaje de error
Fin Clase
```

Código en Java:

Analisis_Sintactico.java

```java
import java.util.Scanner;

public class Analisis_Sintactico {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Ingrese una expresión: ");
        String expresion = scanner.nextLine();

        try {
            Parser parser = new Parser(expresion);
            parser.analizar();
            System.out.println("Expresión válida.");
        } catch (RuntimeException e) {
            System.out.println(e.getMessage());
        }

        scanner.close();
    }
}

/**
 * Clase que implementa el analizador sintáctico descendente recursivo
 */
class Parser {
    private String input;
    private int index = 0;
    private char tokenActual;

    public Parser(String input) {
        this.input = input.replaceAll("\\s+", ""); // Eliminar espacios en blanco
        this.tokenActual = obtenerSiguienteToken();
    }
```

Analisis_Sintactico.java

```java
    private char obtenerSiguienteToken() {
        return (index < input.length()) ? input.charAt(index++) : '\0';
    }

    public void analizar() {
        E();
        if (tokenActual != '\0') {
            error("Expresión mal formada.");
        }
    }

    private void E() {
        T();
        E_();
    }

    private void E_() {
        if (tokenActual == '+' || tokenActual == '-') {
            char operador = tokenActual;
            tokenActual = obtenerSiguienteToken();
            T();
            E_();
        }
    }

    private void T() {
        F();
        T_();
    }

    private void T_() {
        if (tokenActual == '*' || tokenActual == '/') {
            char operador = tokenActual;
```

Analisis_Sintactico.java

Source | History

```
68                  tokenActual = obtenerSiguienteToken();
69                  F();
70                  T_();
71              }
72          }
73
74          private void F() {
75              if (tokenActual == '(') {
76                  tokenActual = obtenerSiguienteToken();
77                  E();
78                  if (tokenActual == ')') {
79                      tokenActual = obtenerSiguienteToken();
80                  } else {
81                      error("Se esperaba ')'.");
82                  }
83              } else if (Character.isDigit(tokenActual)) {
84                  // Manejar números completos
85                  while (Character.isDigit(tokenActual)) {
86                      tokenActual = obtenerSiguienteToken();
87                  }
88              } else if (Character.isLetter(tokenActual)) {
89                  // Manejar identificadores (variables)
90                  tokenActual = obtenerSiguienteToken();
91              } else {
92                  error("Token inesperado.");
93              }
94          }
95
96          private void error(String mensaje) {
97              throw new RuntimeException("Error de sintaxis: " + mensaje);
98          }
99      }
100
```

Resultado de la exprensión en Java:

Output - Analisis_Sintactico (run) | Breakpoints

```
run:
Ingrese una expresi□n: (10+5)/2
Expresi□n v□lida.
BUILD SUCCESSFUL (total time: 14 seconds)
```

Conclusión: La implementación del analizador sintáctico descendente recursivo en Java demuestra un enfoque efectivo para validar y procesar expresiones aritméticas simples. A través de una serie de reglas gramaticales definidas en métodos recursivos, el programa es capaz de descomponer una expresión en sus componentes fundamentales, como términos, factores, operadores y paréntesis, para asegurar que su sintaxis sea correcta. Este tipo de análisis es crucial en compiladores e intérpretes, ya que permite verificar la estructura de las expresiones antes de realizar la ejecución de las operaciones.
La validación se lleva a cabo de manera eficiente, y cualquier error de sintaxis se maneja mediante excepciones que proporcionan mensajes claros sobre la naturaleza del problema. Sin embargo, el algoritmo puede mejorarse para manejar expresiones más complejas o incluir soporte para más tipos de datos y operadores. En resumen, el algoritmo ejemplifica la base de la construcción de un compilador o un evaluador de expresiones, resaltando la importancia de una correcta identificación de los tokens y el uso adecuado de las reglas gramaticales en el análisis de una expresión.

6. Conclusión

El estudio del análisis sintáctico es fundamental para entender cómo los compiladores interpretan y estructuran el código fuente, transformándolo en una representación que pueda ser procesada y ejecutada por una máquina. Este proceso de análisis es crucial, ya que permite garantizar que el código cumpla con las reglas gramaticales de un lenguaje de programación, identificando de manera precisa errores y posibles inconsistencias antes de que el programa sea ejecutado.

El análisis sintáctico se encarga de construir árboles sintácticos, los cuales son representaciones visuales de la jerarquía y la estructura del programa. Estos árboles no solo sirven para organizar el código, sino que también son esenciales para la detección de errores. Al detectar inconsistencias o violaciones en las reglas gramaticales del lenguaje, el análisis sintáctico ayuda a prevenir fallos de ejecución en etapas posteriores, contribuyendo a la robustez y confiabilidad del software. Además, los árboles sintácticos sirven como la base para la generación de código intermedio, una etapa crítica en el proceso de compilación, que facilita la conversión del código de alto nivel a instrucciones de bajo nivel que la máquina pueda ejecutar. Los métodos de análisis sintáctico, como el análisis descendente y ascendente, presentan diferentes enfoques y características, cada uno con sus ventajas y limitaciones. El análisis descendente, por ejemplo, es más intuitivo y fácil de implementar, pero puede tener problemas con gramáticas recursivas por la izquierda, lo que limita su aplicabilidad. En contraste, el análisis ascendente es más eficiente para una gama más amplia de gramáticas, pero su implementación suele ser más compleja y requiere un manejo más cuidadoso de las estructuras intermedias. La elección entre uno u otro depende de las características específicas de la gramática del lenguaje de programación y de las necesidades del compilador.

Comprender las técnicas y herramientas utilizadas en el análisis sintáctico es indispensable para el desarrollo de compiladores eficientes. A medida que los lenguajes de programación evolucionan y se vuelven más complejos, las estrategias de análisis deben adaptarse para mantener la eficiencia y precisión. Además, el análisis sintáctico no solo se limita a la creación de compiladores; también es una herramienta clave en el procesamiento de lenguajes naturales, la verificación formal de programas y la construcción de sistemas inteligentes que manejan lenguajes formales. La optimización de estos procesos no solo mejora la velocidad y la calidad de los compiladores, sino que también impacta en la capacidad de las máquinas para comprender y ejecutar instrucciones de manera más efectiva.

En resumen, el análisis sintáctico es un pilar en la ingeniería de compiladores y en la programación de lenguajes formales. No solo permite transformar el código fuente en una estructura organizada y comprensible, sino que también facilita la identificación de errores y la creación de una versión intermedia del programa, lista para la ejecución. Su estudio profundo es esencial para la creación de herramientas que mejoren la productividad en el desarrollo de software y aseguren la correcta interpretación y ejecución del código en sistemas de computación cada vez más complejos.

7. Referencias Bibliográficas

References

[1] Carraza Sahagún, D. U. (2024) (Coordinador). *Compiladores: fases de análisis.* Editorial Transdigital.https://doi.org/10.56162/transdigitalb44