Universidad Autónoma del Estado de Hidalgo

AUTÓMATAS Y COMPILADORES

Dr. Eduardo Cornejo -Velázquez

NOMBRE DE LA PRÁCTICA:

2.4 Análisis sintáctico. Ejercicios

ALUMNO:

Daniel Martínez Escamilla

Herramientas empleadas

Las herramientas utilizadas para el desarrollo de los ejercicios relacionados con el analizador sintáctico incluyen el libro Çompiladores fase de análisis"del autor Diego Ulises Carranza Sahagún. Este recurso proporciona la base teórica y práctica necesaria para comprender las fases de análisis en la construcción de compiladores y aplicar esos conocimientos en los ejercicios que estás desarrollando. Y el software gratuito CANVA para el desarrrolo de los elementros gráficos y su mayor visibilidad de los procesos

Introducción

Un compilador sintáctico es una parte del compilador que se encarga específicamente de verificar la sintaxis del código. Este toma los tokens generados por el analizador léxico y los organiza en una estructura de árbol llamada árbol sintáctico que pueden ser ascendentes o descendentes, que refleja cómo se deben organizar y procesar las instrucciones. Los compiladores sintácticos utilizan gramáticas formales, como gramáticas libres de contexto, para definir las reglas sintácticas del lenguaje. Estas reglas especifican cómo deben organizarse los tokens en secuencias válidas (por ejemplo, una sentencia ïf.º una expresión aritmética).

Objetivo general

El objetivo de esta práctica es explorar los fundamentos del análisis sintáctico y su aplicación en la construcción de analizadores sintácticos. Se busca que adquieramos habilidades para implementar y entender los procesos que permiten analizar la estructura gramatical de un lenguaje de programación, aplicando técnicas de análisis descendente y ascendente para validar la correcta organización del código fuente.

Objetivos específicos

1.- Comprender los principios fundamentales del análisis sintáctico y su rol en la construcción de compiladores, diferenciando los métodos de análisis como el descendente y ascendente.

2.-Implementar un analizador sintáctico básico utilizando técnicas de análisis de tipo descendente o ascendente para interpretar la estructura gramatical de un lenguaje de programación.

3.-Evaluar la estructura gramatical utilizando herramientas de como el árbol sintáctico, para detectar incoherencias o violaciones a las reglas del lenguaje de programación.

Marco teórico

El análisis sintáctico es una de las fases cruciales en la construcción de compiladores y otros sistemas de procesamiento de lenguajes. Su propósito es examinar el código fuente y verificar si sigue las reglas gramaticales definidas por el lenguaje de programación. En términos sencillos, el análisis sintáctico se encarga de construir una estructura jerárquica que refleje la organización y relación de los elementos dentro del código. Este proceso transforma una secuencia lineal de tokens generada por el análisis léxico en una estructura de datos, generalmente representada como un árbol sintáctico que representa la estructura jerárquica del código y la relación entre sus elementos. Cada nodo del árbol refleja una producción de la gramática, y las hojas corresponden a los tokens del código fuente. Esta representación es útil no solo para la verificación de la corrección gramatical, sino también para la posterior generación de código intermedio o la ejecución de transformaciones del programa.
El análisis de errores sintácticos es también una parte importante del proceso, ya que permite identificar y corregir problemas en la estructura del código. Si el análisis detecta un error, debe informar al programador sobre la naturaleza del mismo, indicando el lugar exacto y el tipo de error cometido, lo que es crucial para la depuración efectiva.

Ejercicio número 1

1. a) Escriba una gramática que genere el conjunto de cadenas
”s; , s;s; , s;s;s; , ...”

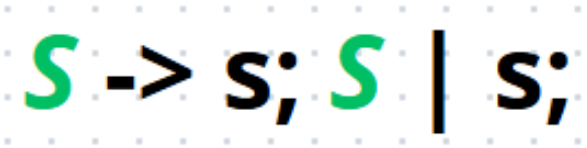

Figura 1: Gramática de la cadena

S → s; S: Permite la concatenación de ”s;çon una secuencia adicional de ”s;”.
S → s;: El caso base, donde la cadena es solo ”s;”.

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

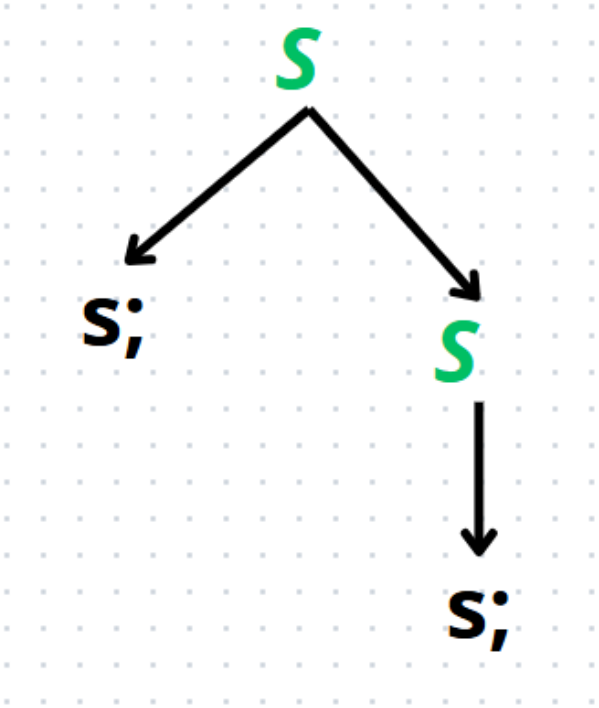

Figura 2: Árbol sintáctico descendente

Este árbol muestra cómo se descompone la cadena ”s;s;.en las producciones de la gramática. Primero se genera el ”s;”de la producción inicial, y luego se aplica la producción S → s; para generar la segunda parte de la cadena.

Ejercicio número 2

Considere la siguiente gramática:
rexp → rexp “|” rexp
— rexp rexp
— rexp “*”
— “(“ rexp “)”
— letra
a) Genere un árbol sintáctico para la expresión regular (ab|b)*.

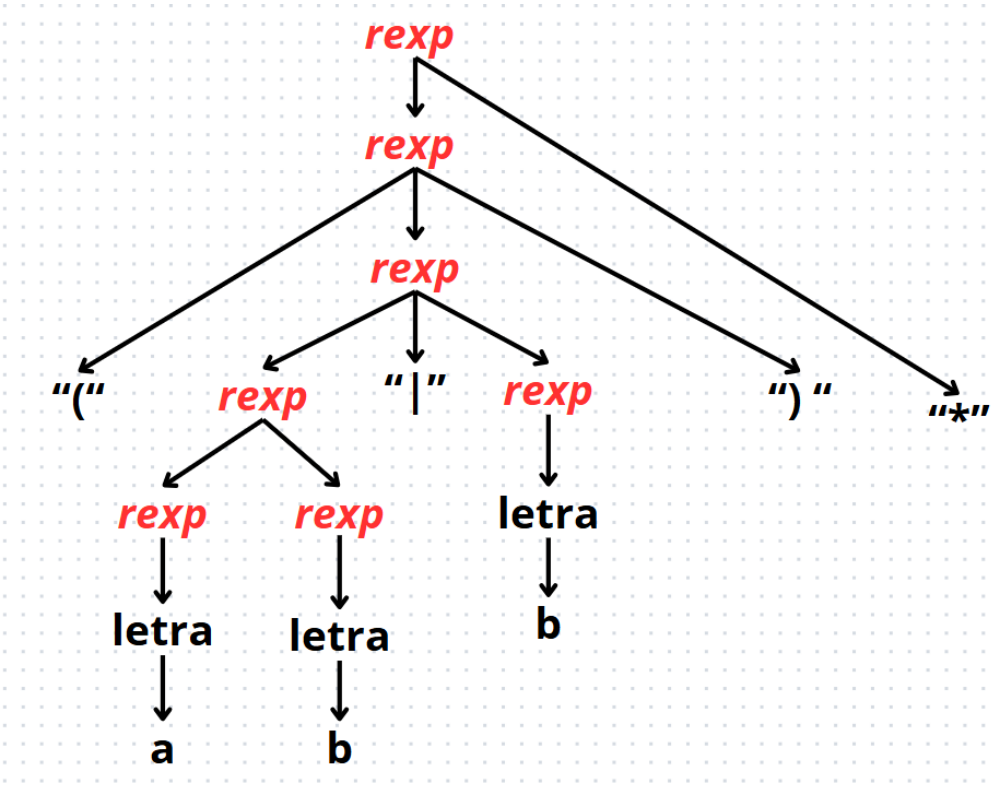

Figura 3: Árbol sintáctico:

Este árbol refleja cómo la expresión regular (ab| b) se descompone de acuerdo con la gramática dada.

Ejercicio número 3

3. 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*.

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

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

a) S → S S + | S S * | a con la cadena aa + a*
Lenguaje generado: {a,aa+,aa+a*...}

La producción S → a genera una sola 'a'.

Las producciones S → S S + y S → S S * permiten la concatenación de dos "S"seguidas de un operador '+' o '*'.

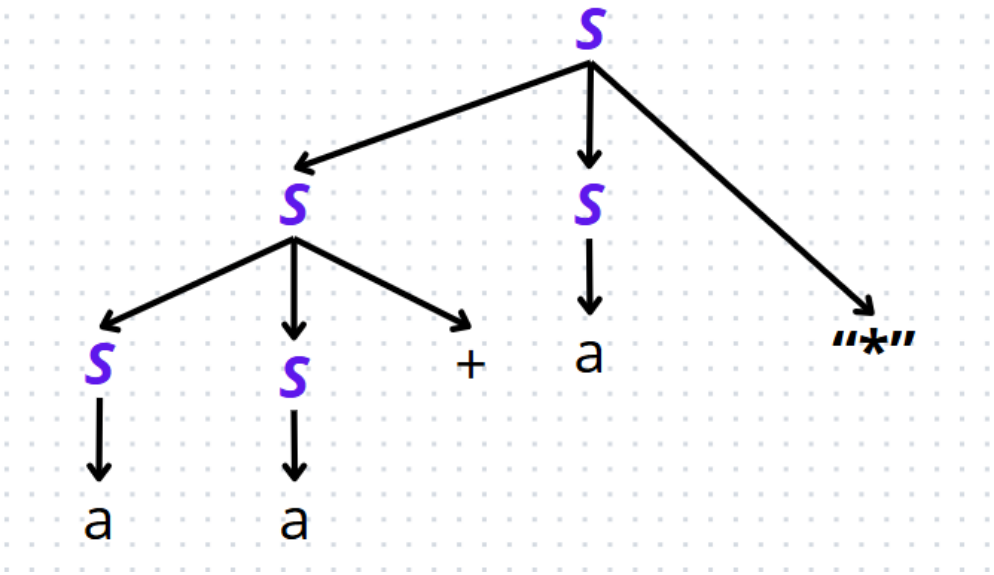

Figura 4: Árbol sintáctico:

b) S → 0 S 1 | 0 1 con la cadena 000111
Lenguaje generado:{01,0011,0011,000111...}

La producción S → 0 S 1 permite generar cadenas que comienzan con '0' y terminan con '1', con una 'S' en el medio.

La producción S → 0 1 genera la secuencia de "01".

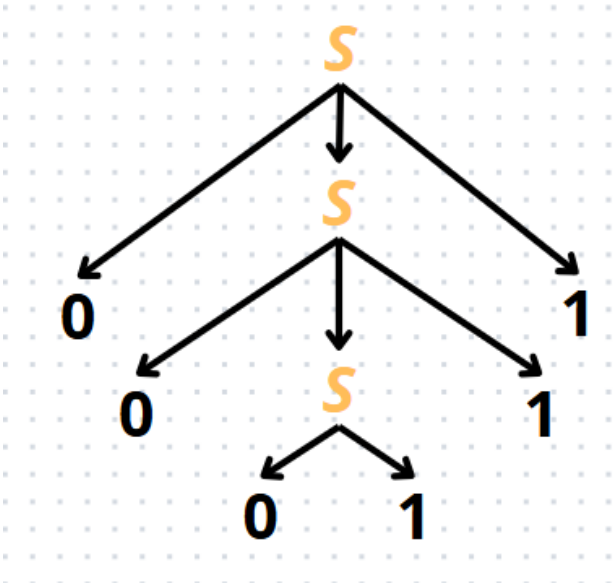

Figura 5: Árbol sintáctico:

c) S → + S S | * S S | a con la cadena + * aaa
Lenguaje generado:{a,*aa,+*aaa,...}

La producción S → a genera 'a'.

La producción S → + S S genera una cadena que comienza con '+' y sigue con dos 'S'.

La producción S → * S S genera una cadena que comienza con '*' y sigue con dos 'S'.

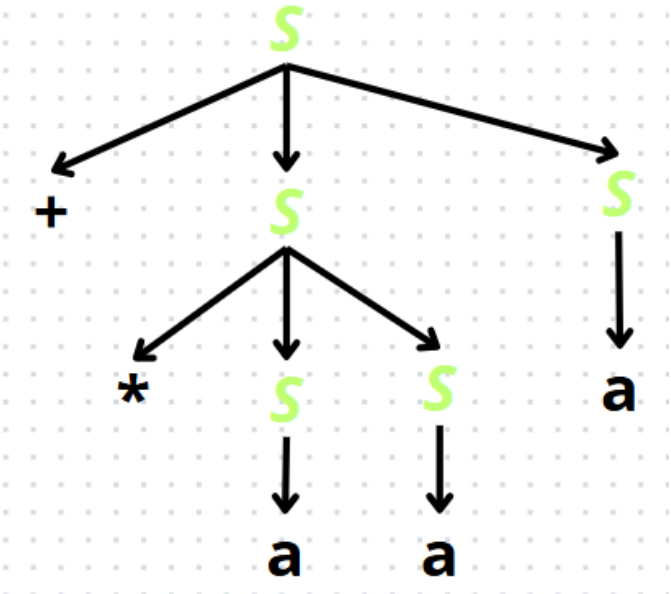

Figura 6: Árbol sintáctico:

Ejercicio número 4

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

S → xSy | e

Producción S → xSy:
Esta producción genera una cadena que empieza con un x, seguida de otra cadena derivada de S, y termina con un y.

Tenemos que:
Si S genera la cadena vacía 'e', entonces la producción S → xSy genera xy.
Si S genera xSy, entonces la producción S → xSy genera xSy → xxSyy de manera 'infinita'.

Lenguaje generado: {xy,xxyy,xxxyyy,...}
El lenguaje generado por esta gramática consiste en todas las cadenas que tienen una cantidad igual de x y y, con todas las x al principio y todas las y al final. La cadena vacía también es aceptada.

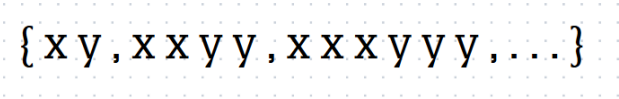

Figura 7: Lenguaje

Ejercicio número 5

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

S → zMNz
M → aNa
N → bNb
N → z

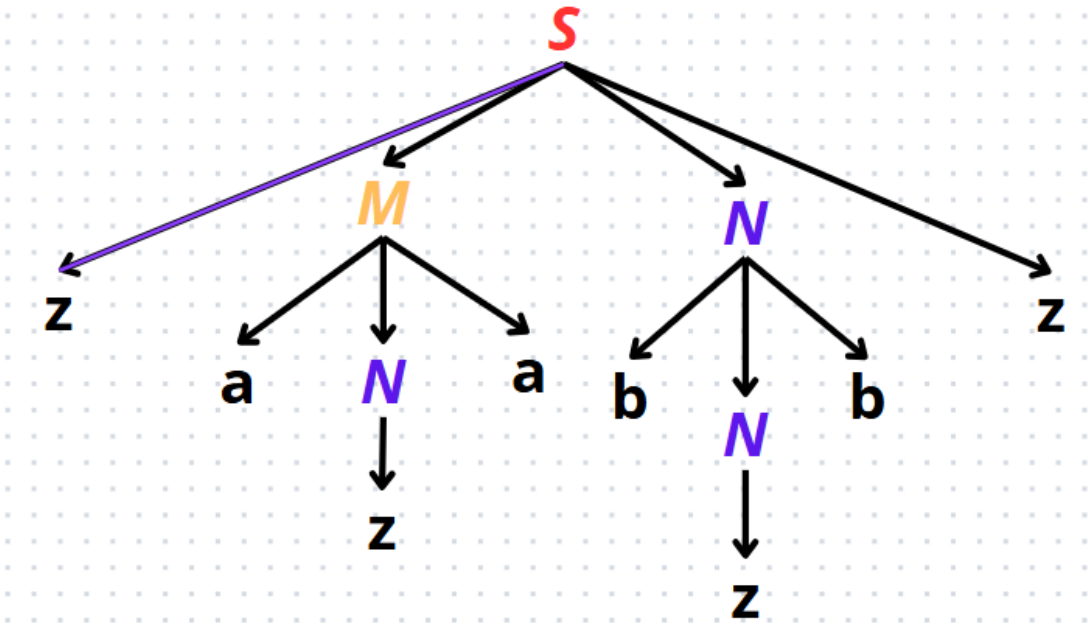

Figura 8: Árbol sintáctico:

La raíz del árbol es S, que se descompone en zMNz.

El nodo M se descompone en aNa.

El nodo N se descompone en bNb, y luego el N se reemplaza por z

Ejercicio número 6

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 → ictS
S → ictSeS
S → s

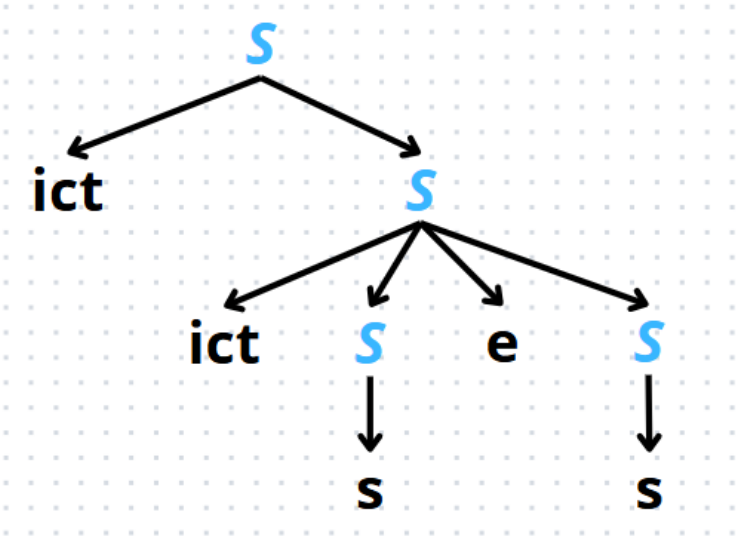

Figura 9: Primera derivación

Otra derivación:

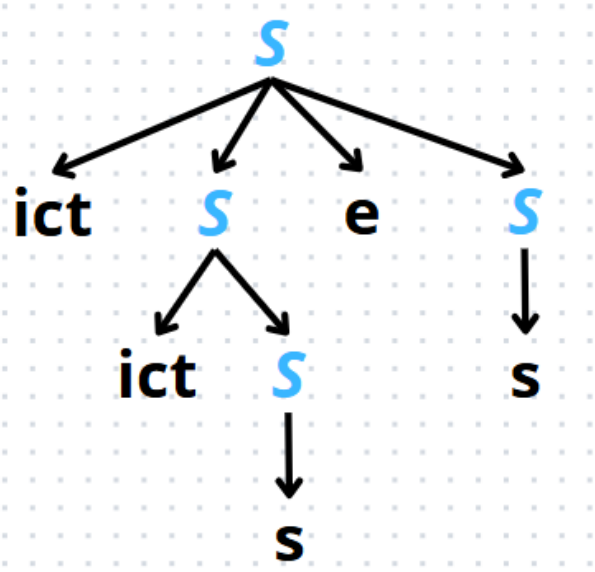

Figura 10: Segunda derivación

Ejercicio número 7

7. Considere la siguiente gramática

S → (L) | a
L → L , S | S

Encuéntrense árboles de análisis sintáctico para las siguientes frases:

a) (a, a)
b) (a, (a, a))
c) (a, ((a, a), (a, a)))

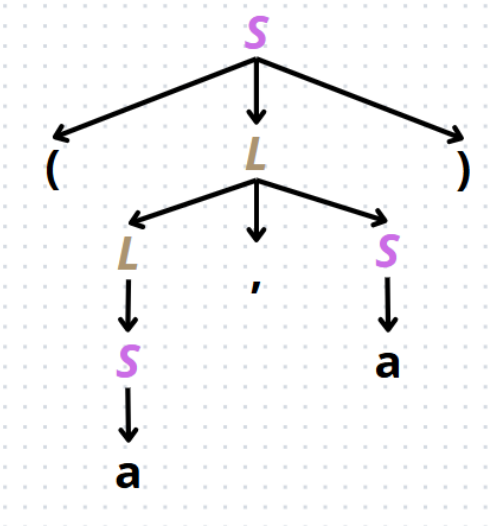

Figura 11: a) (a, a)

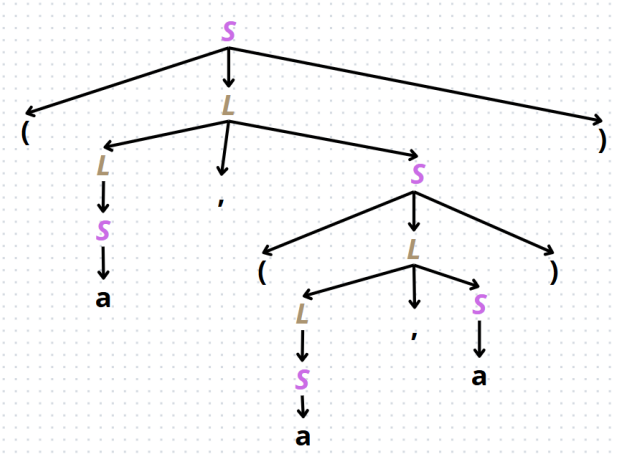

Figura 12: b) (a, (a, a))

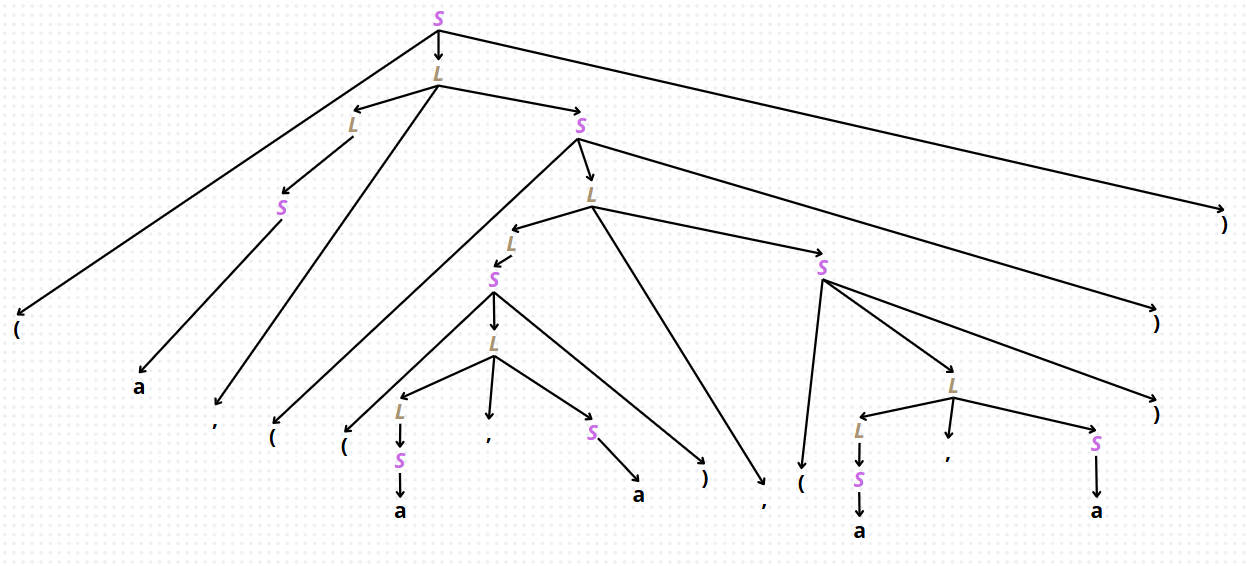

Figura 13: c) (a, ((a, a), (a, a)))

Ejercicio número 8

8. 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

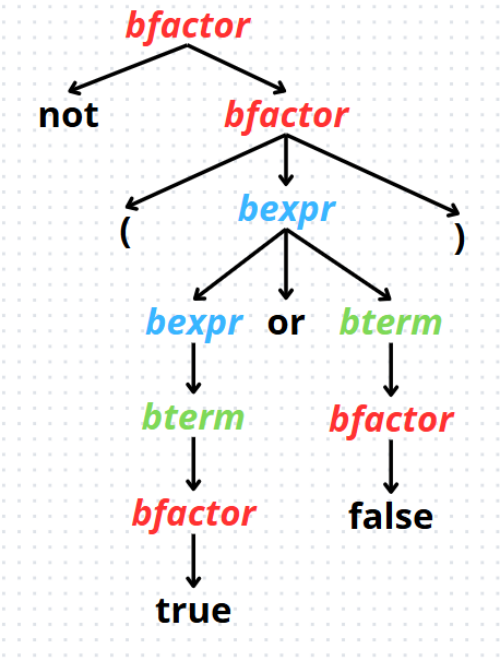

Figura 14: Árbol sintáctico

Ejercicio número 9

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.

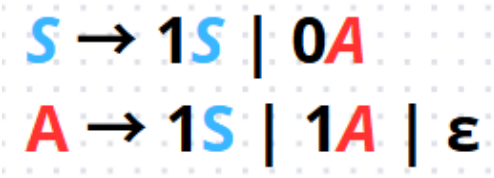

Figura 15: Gramática para el lenguaje:

S → 1S: Esta regla permite que la cadena empiece con un 1 y continúe con cualquier secuencia que siga las reglas de la gramática.

S → 0A: Esta regla permite que un 0 sea seguido de una subcadena que se derive desde A, asegurando que todo 0 esté seguido de al menos un 1.

A → 1S: Si hemos llegado a la parte después de un 0, debemos tener un 1 y luego cualquier secuencia que continúe siguiendo las reglas de S.

A → 1A: Si hemos llegado a la parte después de un 0, debemos tener un 1 y luego cualquier secuencia que continúe siguiendo las reglas de A.

A → ∈: Esto permite que la secuencia termine sin más símbolos después de un 1, lo que significa que un 0 que haya sido seguido de un 1 puede terminar la cadena.

La gramática garantiza que cada 0 esté seguido inmediatamente de un 1, y que la secuencia de 1s y 0s sea válida.

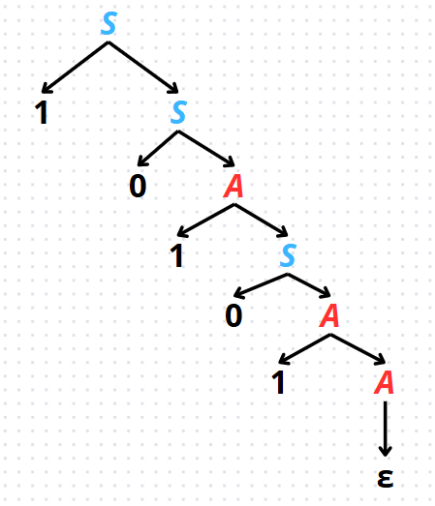

Figura 16: Árbol sintáctico:

Ejercicio número 10

10.Elimine la recursividad por la izquierda de la siguiente gramática:

S → (L) | a
L → L , S | S

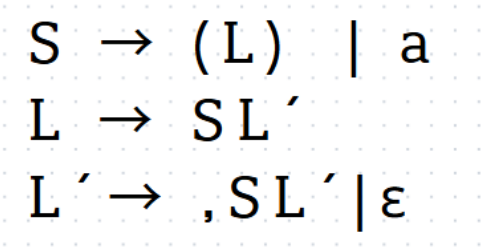

Figura 17: Eliminando Recursividad por la izquierda

Para S: No es recursiva por la izquierda por lo que no se modifica

Para L: La producción L → L , S | S se ha dividido en dos partes. La primera parte L → S L' asegura que L comience con S, seguido de la nueva variable L', que maneja la continuación de la lista.

Para L': La nueva producción L' → , S L' | ∈ permite que la lista de elementos continúe con un , S, o termine de manera vacía con ∈ lo que permite que L también pueda ser simplemente S sin más elementos.

Ejercicio número 11

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

Descripción del método descendente recursivo:
El análisis descendente recursivo utiliza una función recursiva para analizar cada no terminal en la gramática. Para esta gramática, tenemos las siguientes reglas de producción:

S → (S): Esto implica que S puede ser una cadena de paréntesis que contiene otro S dentro de ellos.

S → x: Esto significa que S puede ser simplemente el símbolo x.

Pseudocódigo para el análisis sintáctico:
El pseudocódigo seguirá una estructura donde, en función del símbolo que leemos, se toma la producción adecuada. La función recursiva intentará descomponer la cadena de entrada según las reglas de la gramática.

```
FUNCION analizar_S():
   SI el siguiente símbolo es "(":
      Avanzar en la entrada (consumir "(")
      Llamar recursivamente a analizar_S()
      SI el siguiente símbolo es ")":
         Avanzar en la entrada (consumir ")")
         Retornar éxito
      SINO:
         Retornar error (esperaba ")")
   SINO SI el siguiente símbolo es "x":
      Avanzar en la entrada (consumir "x")
      Retornar éxito
   SINO:
      Retornar error (símbolo inesperado)

INICIO
   Llamar a analizar_S()
   SI la entrada está vacía:
      Retornar éxito (cadena válida)
   SINO:
      Retornar error (entrada incompleta)
FIN
```

Figura 18: Pseudocódigo:

Explicación:
Función analizar..S: Esta función intenta analizar el símbolo S en función del símbolo siguiente en la entrada.

Si el siguiente símbolo es ”(”, entonces la función consume ese símbolo, llama recursivamente a analizar..S para analizar el contenido entre los paréntesis, y luego verifica si el siguiente símbolo es ”)”.

Si el siguiente símbolo es ”x”, la función simplemente consume ese símbolo y termina con éxito.

Si el siguiente símbolo no es ni (ni x, se reporta un error.

Estructura general: En el cuerpo principal, se llama a analizar..S al principio para comenzar el análisis. Si se llega al final de la cadena sin errores, la entrada es válida; si no, se indica un error.

Ejercicio número 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 → 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*)		

Figura 19: Tabla 3.1

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'**

Figura 20: Parte 1

$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()

Figura 21: Parte 2

Ejercicio número 13

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

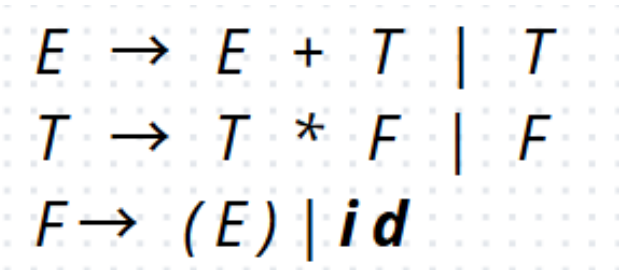

Figura 22: Gramática original

Hacemos las respectivas modificaciones a la gramática original para que nos acepte las operaciones de resta (-) y la de división (/) y pueda derivar la ultima producción en 'num':
Primero haremos la modificación en la primer producción:

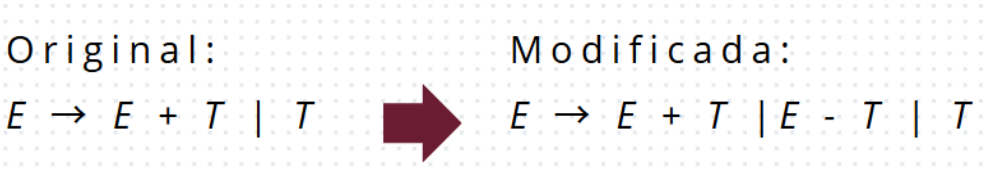

Figura 23: Primera producción modificada

Posterioirmente la modificación en la segunda producción:

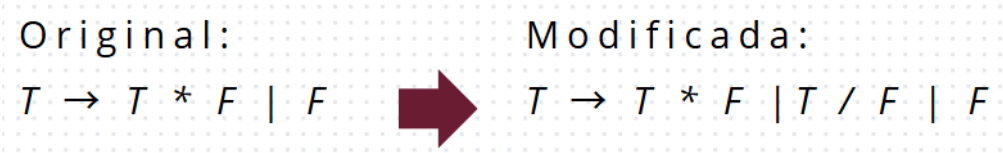

Figura 24: Segunda producción modificada

Por último modificaremos la ultíma producción agregandole la derivaación 'num':

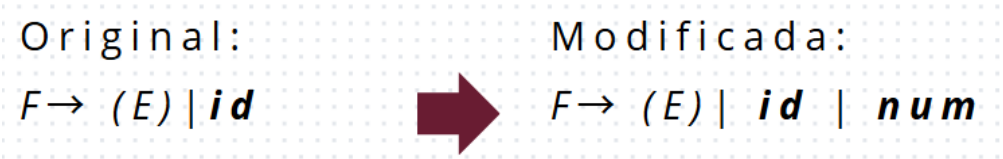

Figura 25: Última producción modificada

Y aquí tenemos la gramática general con sus respectivas modificaciones:

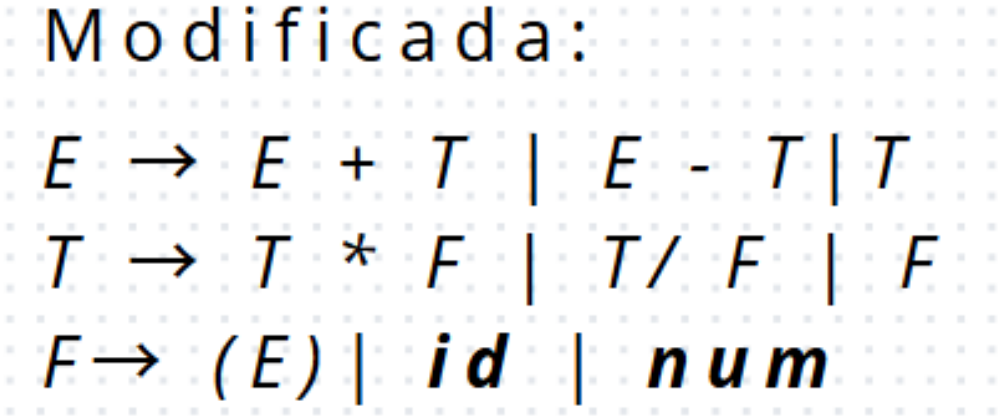

Figura 26: Gramática modificada

Posteriormente a nuestra gramática ya modificada le realizamos la eliminación de recursividad por la izquierda:

Figura 27: Primera producción sin recursividad por la izquierda

Hacemos la eliminación de recurisividad a la segunda produccón:

Figura 28: Segunda producción sin recursividad por la izquierda

Por último incorporamos todas las producciones en la misma gramática y tenemos como resultado la nueva gramática sin recursividad por la izquierda:

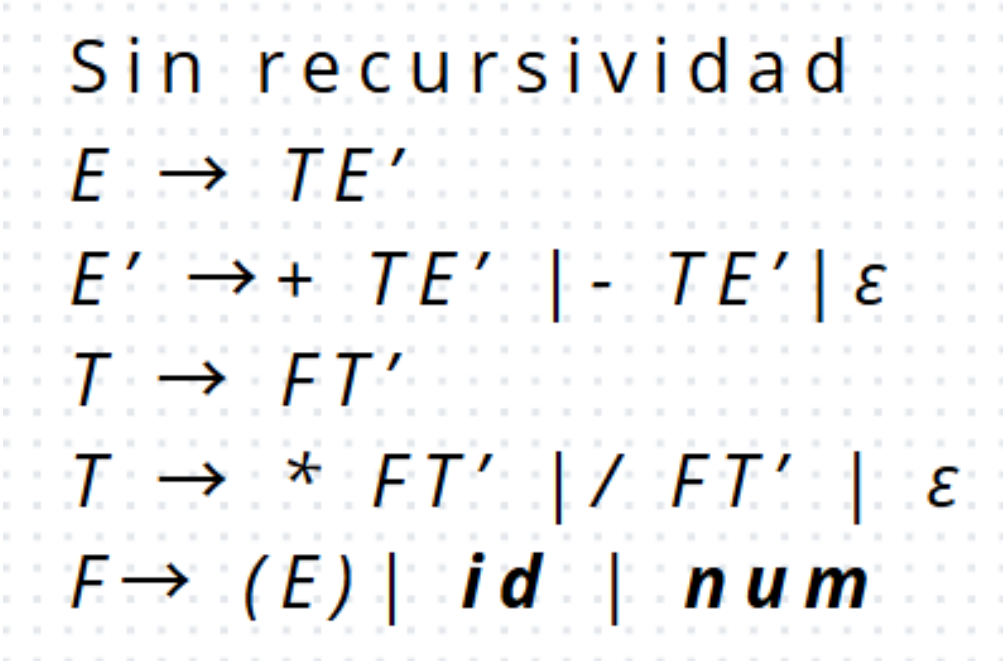

Figura 29: Gramática general sin recursividad por la izquierda

Como podemos observar tenemos que la última producción no recibió algún ca bio ya que en esta no hay recursividad ya que ningún símbolo de la izquierda se repite en la derecha.

Ejercicio número 14

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

Primera parte. El pseudocódigo

```
FUNCIÓN E():
    T()
    E'()

FUNCIÓN E'():
    SI token_actual es '+' o token_actual es '-':
        consumir(token_actual)  // Consume el operador '+ o -'
        T()
        E'()  // Llamada recursiva a E'
    SINO SI token_actual es vacío (ε):
        //  Epsilon implica que no hay más producción

FUNCIÓN T():
    F()
    T'()

FUNCIÓN T'():
    SI token_actual es '*' o token_actual es '/':
        consumir(token_actual)  // Consume el operador '*' o '/'
        F()
        T'()  // Llamada recursiva a T'
    SINO SI token_actual es vacío (ε):
        // No hace nada

FUNCIÓN F():
    SI token_actual es '(':
        consumir(token_actual)
        E()  // Procesa la expresión interna
        SIi token_actual es ')':
            consumir(token_actual)  // Consume el paréntesis ')'
    SINO SI token_actual es 'id':
        consumir(token_actual)  // Consume un identificador
    SINO SI  token_actual es 'num':
        consumir(token_actual)  // Consume un número
    SINO:
        //Entra algin token inesperado
        mostrar_error()

FUNCIÓN consumir(token):
    // Avanza al siguiente token en la entrada
    token_actual = siguiente_token()
```

Figura 30: Pseudocódigo para solución de ejercicio 13

Explicación del pseudocódigo:
E(): Procesa una expresión, que está formada por un término seguido de un .E'"que puede contener más operaciones (suma o resta).

E'(): Si el token actual es un operador de suma o resta, lo consume y luego llama recursivamente a T() para procesar el siguiente término y a E'() para manejar más operadores. Si no hay más operadores (e), simplemente retorna sin hacer nada.

T(): Procesa un término, que está formado por un factor seguido de un "T""que maneja las multiplicaciones o divisiones.

T'(): Si el token actual es un operador de multiplicación o división, lo consume y luego llama a F() para procesar el siguiente factor y a T'() para manejar más operadores. Si no hay más operadores (e), simplemente retorna sin hacer nada.

F(): Procesa un factor. El factor puede ser una expresión entre paréntesis, un identificador o un número.

Implementación del pseudocódigo en JAVA:

```java
package Prueba1;
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Scanner;

public class AS_1 {


    private List<String> tokens;
    private int posicion = 0;
    private String tokenAct;


    public AS_1(String expresion) {
        // Convierte la expresión en una lista de tokens
        tokens = new ArrayList<>(c: Arrays.asList(a: expresion.split(regex: " ")));
        sigToken();
    }

    // Función principal para analizar la expresión
    public void analizar() {
        E();
        if (posicion < tokens.size()) {
            System.out.println(x: "Error de sintaxis: Tokens adicionales no esperados.");
        } else {
            System.out.println(x: "La expresión es válida.");
        }
    }

    // E → TE'
    private void E() {
        T();
        Ep();
    }
```

Figura 31: Implementación del pseudocódigo en JAVA

```java
// E' → + TE' | -TE' | ε
private void Ep() {
    if (tokenAct.equals(anObject: "+")|| tokenAct.equals(anObject: "-")) {
        consumir(esperado: tokenAct);  // Consume '+' o '-'
        T();
        Ep();  // Llamada recursiva para procesar más operadores
    }
    // Epsilon es implícito y no hace nada
}

// T → FT'
private void T() {
    F();
    Tp();
}

// T' → * FT' | / FT' | ε
private void Tp() {
    if (tokenAct.equals(anObject: "*")|| tokenAct.equals(anObject: "/")) {
        consumir(esperado: tokenAct);  // Consume '*' o '/'
        F();
        Tp();  // Llamada recursiva
    }
    // Epsilon es implícito y no hace nada
}

// F → (E) | id | num
private void F() {
    if (tokenAct.equals(anObject: "(")) {
        consumir(esperado: "(");  // Consume el paréntesis '('
        E();  // Procesa la expresión interna
        if (tokenAct.equals(anObject: ")")) {
            consumir(esperado: ")");  // Consume el paréntesis ')'
        } else {
            error(mensaje: "Se esperaba ')'");
```

Figura 32: Implementación del pseudocódigo en JAVA

```java
        }
    } else if (tokenAct.equals(anObject: "id")|| tokenAct.equals(anObject: "num")) {
        consumir(esperado: tokenAct);  // Consume un identificador o número
    } else {
        error("Token inesperado: " + tokenAct);
    }
}

// Método para consumir el token actual y avanzar al siguiente
private void consumir(String esperado) {
    if (tokenAct.equals(anObject: esperado)) {
        sigToken();
    } else {
        error("Se esperaba: " + esperado + ", pero se encontró: " + tokenAct);
    }
}

// Método para obtener el siguiente token de la lista
private void sigToken() {
    if (posicion < tokens.size()) {
        tokenAct = tokens.get(index: posicion);
        posicion++;
    } else {
        tokenAct = "";
    }
}

// Método para mostrar los errores
private void error(String mensaje) {
    System.out.println("Error de sintaxis: " + mensaje);
    System.exit(status:1);
}


public static void main(String[] args) {
```

Figura 33: Implementación del pseudocódigo en JAVA

Figura 34: Implementación del pseudocódigo en JAVA

Conclusiones

El análisis sintáctico es una etapa esencial en la construcción de compiladores, ya que garantiza que el código fuente siga las reglas gramaticales del lenguaje de programación. A través de métodos como el análisis descendente y ascendente, que facilita la validación de la correcta organización de los elementos del programa.Por otro lado conocimos algunos de los retos que se llegan a presentar en este análisis, como lo fue la ambiguedad, la recursividad por la izquierda y la factorización. Además, el análisis sintáctico juega un papel crucial en la detección de errores, proporcionando retroalimentación valiosa para la depuración del código. La correcta implementación de esta fase permite que el compilador pueda interpretar adecuadamente la lógica del programa, sentando las bases para las siguientes fases de compilación o interpretación, como la verificación semántica y la generación de código ejecutable. Sin una adecuada capacidad de análisis sintáctico, el desarrollo de compiladores y sistemas de procesamiento de lenguajes sería inviable.

Referencias

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