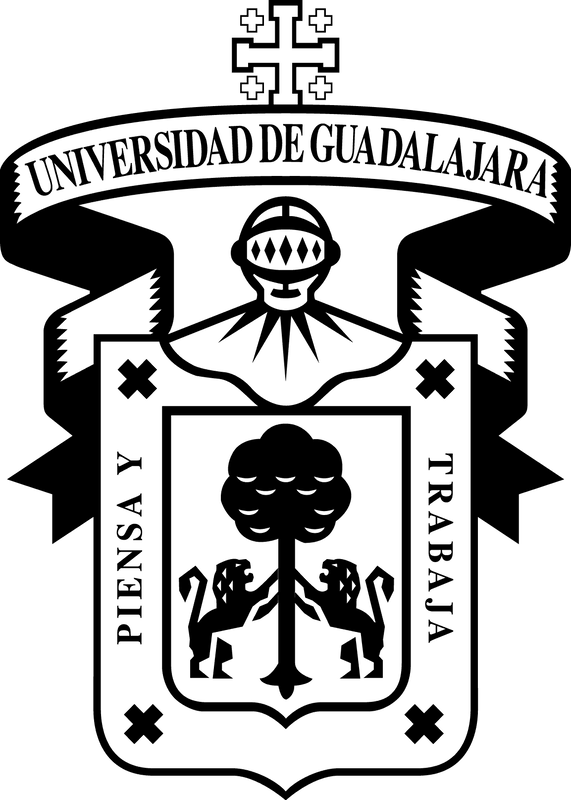
Seminario de Solución de problemas de Traductores de Lenguajes II

# Centro Universitario de Ciencias Exactas en ingenierías

# Universidad de Guadalajara



Maestro: [LUIS FELIPE MUNOZ MENDOZA](mailto:luis.munoz.m@academicos.udg.mx)

Juan Antonio Pérez Juárez

Código: 215660996

Carrera: INCO

Práctica 2 - Analizador Sintáctico

Introducción:

Realizar un analizador sintáctico que reconozca sentencias de asignación u operaciones simples, y valide su estructura con una gramática como la siguiente (ejemplo):

<program> -> <assignment> | <assignment> <program>

<assignment> -> <identifier> = <expression> ;

<identifier> -> [a-zA-Z][a-zA-Z0-9\_]\*

<expression> -> <term> | <term> + <expression> | <term>

* <expression>  
  <term> -> <factor> | <factor> \* <term> | <factor> / <term>  
  <factor> -> <identifier> | <number> | ( <expression> )  
  <number> -> [0-9]+

Desarrollo:

Primero debemos definir lo que es un Analizador sintáctico.

Un analizador sintáctico (parser) o simplemente analizador es un programa informático que analiza una cadena de símbolos según las reglas de una gramática formal. El término proviene del latín pars, que significa parte (del discurso). Usualmente hace uso de un compilador, en cuyo caso, transforma una entrada en un árbol sintáctico de derivación.

El análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada. Un analizador léxico crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico para construir la estructura de datos, por ejemplo un árbol de análisis o árboles de sintaxis abstracta.

El lenguaje natural. Es usado para generar diagramas de lenguajes que usan flexión gramatical, como los idiomas romances o el latín. Los lenguajes habitualmente reconocidos por los analizadores sintácticos son los lenguajes libres de contexto. Cabe notar que existe una justificación formal que establece que los lenguajes libres de contexto son aquellos reconocibles por un autómata de pila, de modo que todo analizador sintáctico que reconoce un lenguaje libre de contexto es equivalente en capacidad computacional a un autómata de pila.

Los analizadores sintácticos fueron extensivamente estudiados durante los años 1970, detectando numerosos patrones de funcionamiento en ellos, cosa que permitió la creación de programas generadores de analizadores sintácticos a partir de una especificación de la sintaxis del lenguaje en forma Backus-Naur por ejemplo, tales como yacc, GNU bison y javaCC.

El uso más común de los analizadores sintácticos es como parte de la fase de análisis de los compiladores. De modo que tienen que analizar el código fuente del lenguaje. Los lenguajes de programación tienden a basarse en gramáticas libres de contexto, debido a que se pueden escribir analizadores rápidos y eficientes para estas.

Las gramáticas libres de contexto tienen una expresividad limitada y sólo pueden expresar un conjunto limitado de lenguajes. Informalmente la razón de esto es que la memoria de un lenguaje de este tipo es limitada, la gramática no puede recordar la presencia de una construcción en una entrada arbitrariamente larga y esto es necesario en un lenguaje en el que por ejemplo una variable debe ser declarada antes de que pueda ser referenciada. Las gramáticas más complejas no pueden ser analizadas de forma eficiente. Por estas razones es común crear un analizador permisivo para una gramática libre de contexto que acepta un superconjunto del lenguaje (acepta algunas construcciones inválidas), después del análisis inicial las construcciones incorrectas pueden ser filtradas.

Normalmente es fácil definir una gramática libre de contexto que acepte todas las construcciones de un lenguaje pero por el contrario es prácticamente imposible construir una gramática libre de contexto que admita solo las construcciones deseadas. En cualquier caso la mayoría de analizadores no son construidos a mano sino usando generadores automáticos.

Habiendo definido lo que es y lo que hace un analizador sintáctico, vayamos al código del que desarrollamos para la práctica 2.

Código:

import re

import sys

class SyntaxParser:

def \_\_init\_\_(self, input\_string):

# Eliminar espacios en blanco

self.tokens = input\_string.replace(' ', '').split(';')

# Eliminar cadenas vacías

self.tokens = [token for token in self.tokens if token]

self.current\_token = None

self.token\_index = 0

def parse(self):

"""

Método principal para parsear el programa completo

"""

try:

while self.token\_index < len(self.tokens):

# Parsear cada instrucción

current\_statement = self.tokens[self.token\_index]

self.token\_index += 1

# Reiniciar índices para cada instrucción

self.current\_token = None

self.pos = 0

# Analizar la instrucción

print(f"Analizando expresion: {current\_statement}")

# Realizar validaciones

resultado = self.validate\_assignment(current\_statement)

# Imprimir resultados detallados

if resultado:

print("--- Validacion Completa ---")

print(f"Identificador: {resultado['identificador']}")

print(f"Expresion: {resultado['expresion']}")

print("Estado: Expresion Sintacticamente Correcta")

return True

except SyntaxError as e:

print(f"Error de sintaxis: {e}")

return False

def validate\_assignment(self, statement):

"""

Validar una sentencia de asignación

"""

# Dividir la asignación en identificador y expresión

parts = statement.split('=')

if len(parts) != 2:

raise SyntaxError("Formato de asignacion invalido")

identificador = parts[0].strip()

expresion = parts[1].strip()

# Validar identificador

if not re.match(r'^[a-zA-Z][a-zA-Z0-9\_]\*$', identificador):

raise SyntaxError(f"Identificador invalido: {identificador}")

# Validar la expresión

self.validate\_expression(expresion)

# Retornar detalles de la validación

return {

'identificador': identificador,

'expresion': expresion

}

def validate\_expression(self, expresion):

"""

Validar una expresión

"""

# Método para dividir la expresión en términos

def split\_by\_operator(expr, operators):

for op in operators:

if op in expr:

parts = expr.split(op, 1)

return parts[0].strip(), op, parts[1].strip()

return expr, None, None

# Registro de operaciones encontradas

operaciones\_encontradas = []

# Primero intentar dividir por suma

term1, op\_suma, term2 = split\_by\_operator(expresion, ['+'])

# Validar el primer término

self.validate\_term(term1)

# Si hay suma, validar el segundo término y registrar

if op\_suma:

operaciones\_encontradas.append('+')

self.validate\_term(term2)

return operaciones\_encontradas

def validate\_term(self, termino):

"""

Validar un término

"""

# Método para dividir el término

def split\_by\_operator(expr, operators):

for op in operators:

if op in expr:

parts = expr.split(op, 1)

return parts[0].strip(), op, parts[1].strip()

return expr, None, None

# Registro de operaciones encontradas

operaciones\_encontradas = []

# Primero intentar dividir por multiplicación o división

factor1, op\_mult, factor2 = split\_by\_operator(termino, ['\*', '/'])

# Validar el primer factor

self.validate\_factor(factor1)

# Si hay multiplicación o división, validar el segundo factor y registrar

if op\_mult:

operaciones\_encontradas.append(op\_mult)

self.validate\_factor(factor2)

return operaciones\_encontradas

def validate\_factor(self, factor):

"""

Validar un factor

"""

# Quitar paréntesis si los hay

if factor.startswith('(') and factor.endswith(')'):

factor = factor[1:-1].strip()

# Recursivamente validar la expresión dentro de los paréntesis

self.validate\_expression(factor)

else:

# Verificar si es un identificador o un número

if re.match(r'^[a-zA-Z][a-zA-Z0-9\_]\*$', factor):

print(f" - Factor Identificador: {factor}")

return "identificador"

elif re.match(r'^[0-9]+$', factor):

print(f" - Factor Numero: {factor}")

return "numero"

else:

raise SyntaxError(f"Factor invalido: {factor}")

def main():

print("Analizador Sintactico de Expresiones")

print("Ingrese una expresion de asignacion (ej. x = 10; o result = (a + b) \* c;)")

print("Presione Enter sin escribir nada para salir.")

while True:

# Solicitar entrada al usuario

entrada = input("\nIngrese su expresion: ")

# Salir si no hay entrada

if not entrada:

print("Saliendo del programa.")

break

# Crear y ejecutar el parser

parser = SyntaxParser(entrada)

parser.parse()

if \_\_name\_\_ == "\_\_main\_\_":

main()



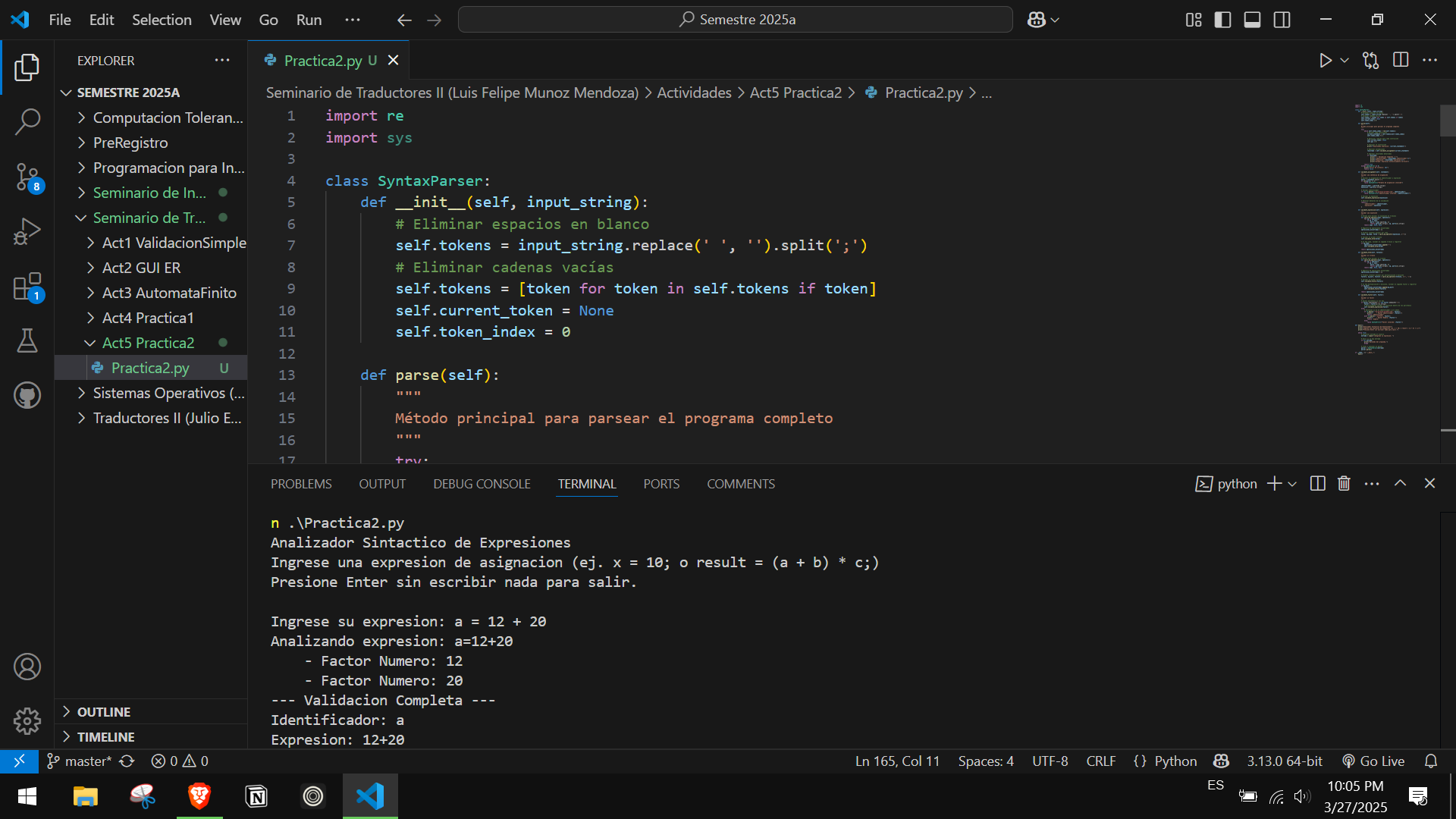
Lógica:

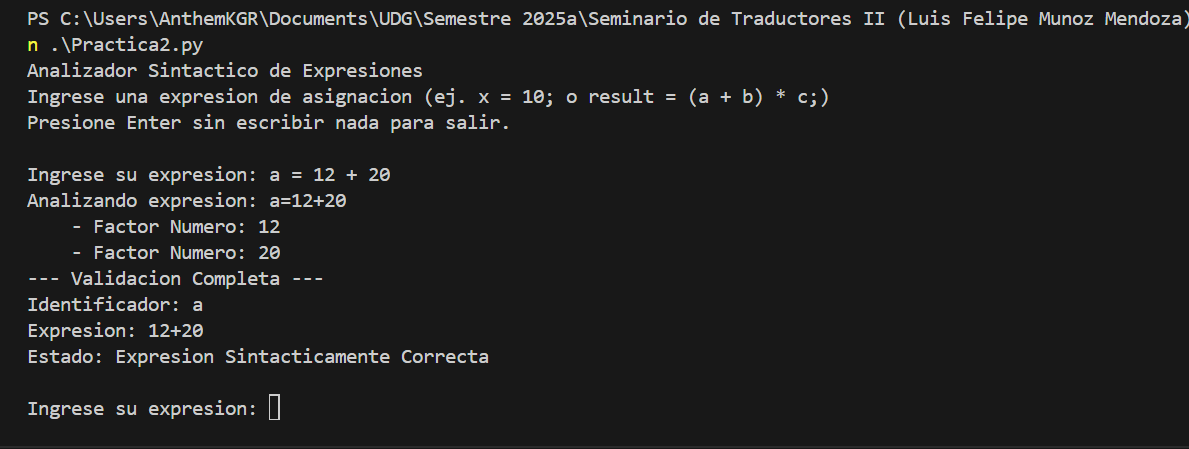
Pasos de validación:

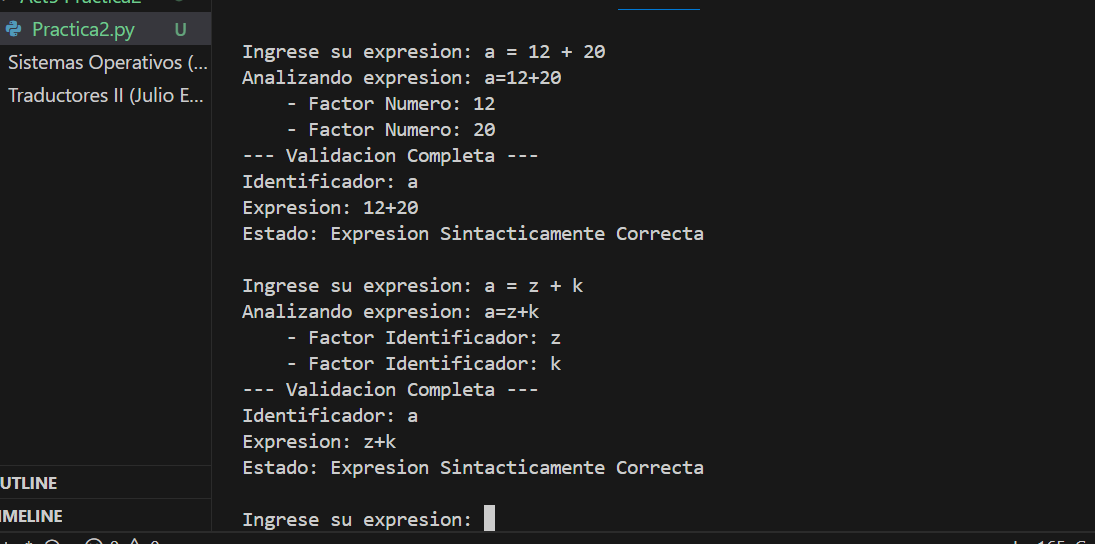
* Divide la asignación en partes
* Verifica el formato correcto
* Valida el identificador con una regex
* Llama al siguiente nivel de parseo
* Función interna para dividir por operadores
* Separa la expresión en términos
* Parsea recursivamente cada término
* Paréntesis: Se eliminan y se parsea recursivamente su contenido

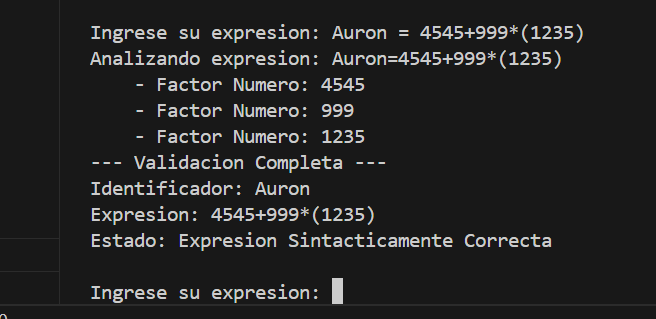
Pruebas:

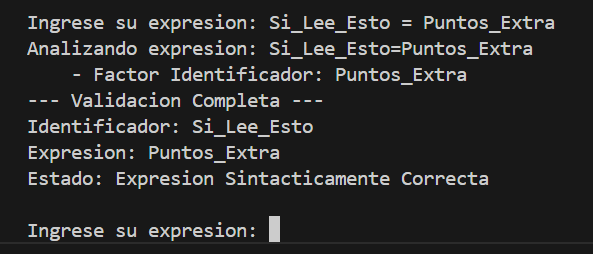
Con una Cadena válida:



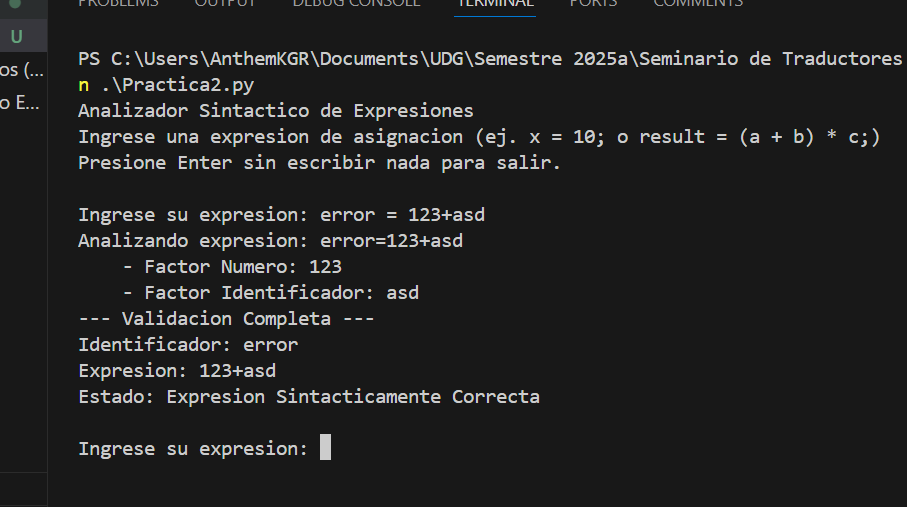


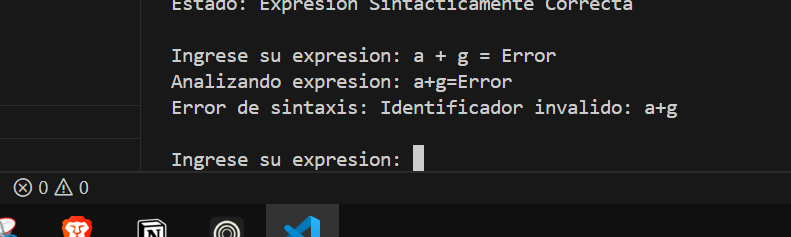






Con una cadena inválida:





Conclusión:

Uffff que difícil la verdad, desde que estamos a mitad de semestre y no me puedo poner a estudiar las presentaciones de la clase, además de darle un repaso a las actividades de teoría de la computación.

Así que he echado mano de inteligencias artificiales para desarrollar la lógica de programación y para que me ayudasen con alguna que otra cosa que no me salía.  
Pero en general esta actividad me gustó, fue como un reto personal, pero pude hacerlo al final para aprender más a fondo el funcionamiento de los compiladores. Lo que me podría ayudar a intentar hacer algo que quiero en el futuro que es hacer un pequeño desarrollo de kernel de Linux.

Referencias:

colaboradores de Wikipedia. (2025, March 22). Analizador sintáctico. Wikipedia, La Enciclopedia Libre. <https://es.wikipedia.org/wiki/Analizador_sint%C3%A1ctico>

argparse — Analizador sintáctico (Parser) para las opciones, argumentos y sub-comandos de la línea de comandos — documentación de Python - 3.10.16. (n.d.). <https://docs.python.org/es/3.10/library/argparse.html>

Analisis-lexico-sintactico-Python/analizador\_lexico.py at master · maryito/Analisis-lexico-sintactico-Python. (n.d.). GitHub. <https://github.com/maryito/Analisis-lexico-sintactico-Python/blob/master/analizador_lexico.py>

parser — Acceder a árboles de análisis sintáctico de Python — documentación de Python - 3.9.21. (n.d.). <https://docs.python.org/es/3.9/library/parser.html>