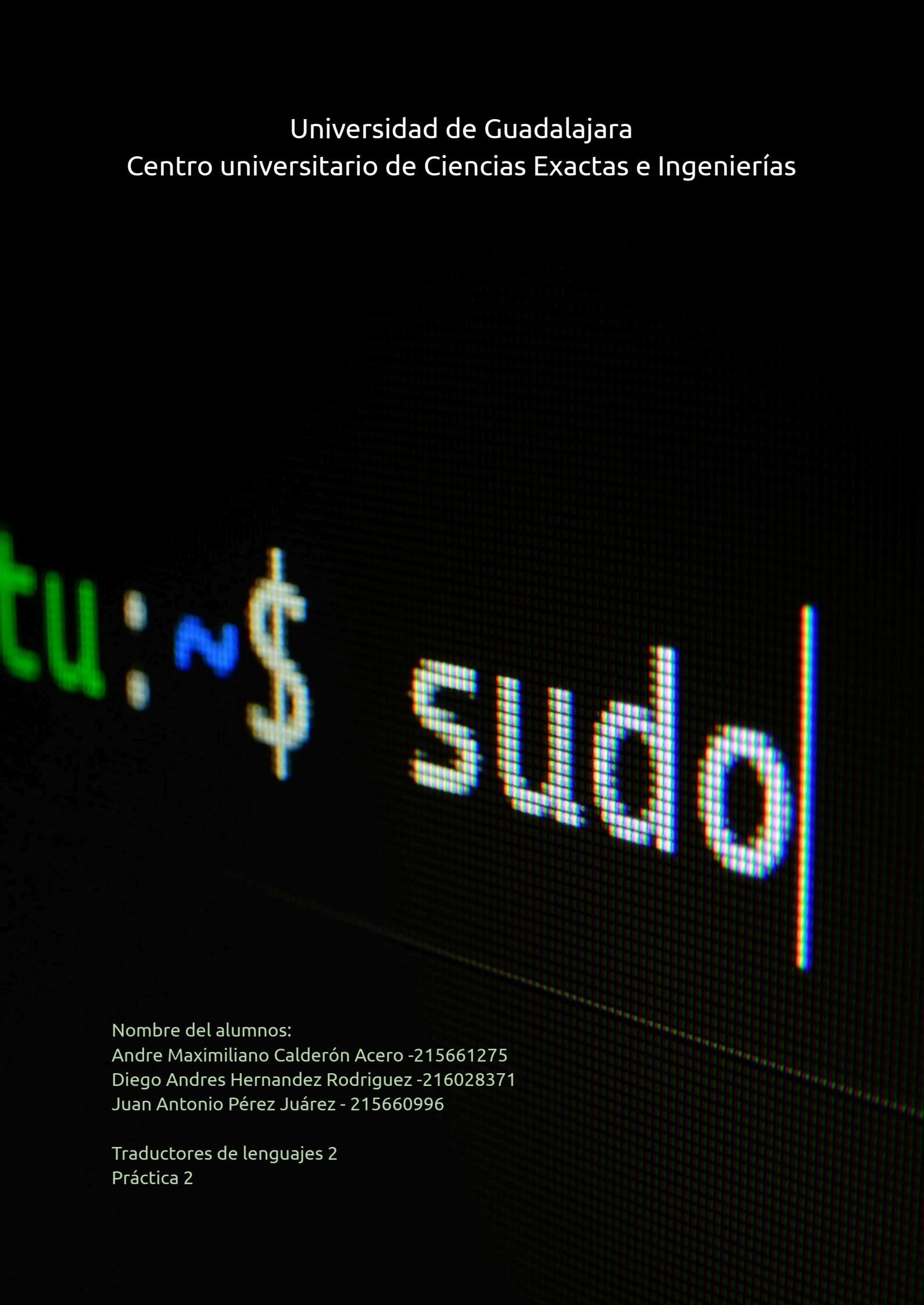
Universidad de Guadalajara
Centro universitario de Ciencias Exactas e Ingenierías



Nombre del alumnos:
Andre Maximiliano Calderón Acero -215661275
Diego Andres Hernandez Rodriguez -216028371
Juan Antonio Pérez Juárez - 215660996

Traductores de lenguajes 2
Práctica 2

Actividad

Implementar un analizador sintáctico descendente con la gramática que reconoce expresiones aritméticas con 4 operaciones y paréntesis.

Si la cadena pertenece al lenguaje se debe mostrar la misma expresión en notación postfija.

Marco Teórico

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.

La tarea esencial de un analizador es determinar si una determinada entrada puede ser derivada desde el símbolo inicial, usando las reglas de una gramática formal, y como hacer esto, existen esencialmente dos formas:

Analizador sintáctico descendente **(Top-Down-Parser)**: ..un analizador puede empezar con el símbolo inicial e intentar transformarlo en la entrada, intuitivamente esto sería ir dividiendo la entrada progresivamente en partes cada

vez más pequeñas, de esta forma funcionan los analizadores LL, un ejemplo es el javaCC. Una mejora en estos parsers se puede logar usando GLR (Generalized Left-to-right Rightmost derivation).

Analizador sintáctico ascendente **(Bottom-Up-Parser)**: un analizador puede empezar con la entrada e intentar llegar hasta el símbolo inicial, intuitivamente el analizador intenta encontrar los símbolos más pequeños y progresivamente construir la jerarquía de símbolos hasta el inicial, los analizadores LR funcionan así y un ejemplo es el Yacc. También existen SLR (Simple LR) o los LALR (Look-ahead LR) como también de los GLL7 (Generalized Left-to-right Leftmost derivation).

Código:

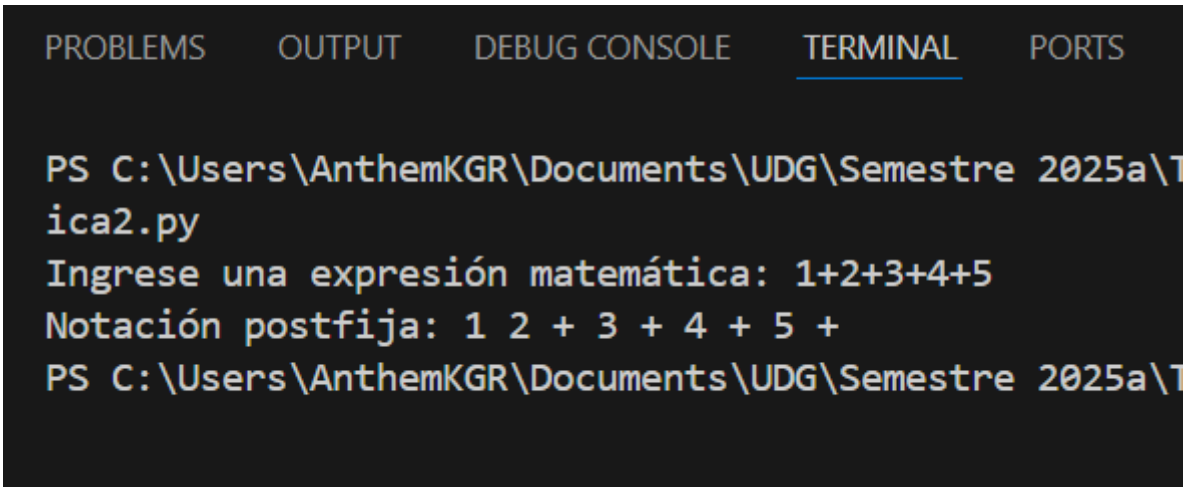
Python

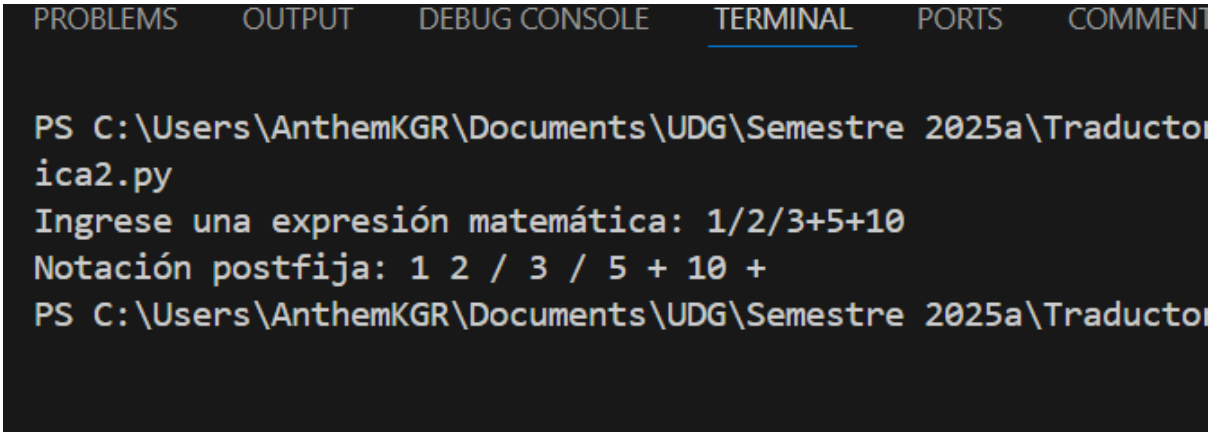
```python
"""Código creado por Diego Andres Hernandez"""
class Parser:
    def __init__(self, expression):
        self.expression = expression.replace(" ", "")  # Eliminamos espacios
        self.index = 0

    def parse(self):
        self.index = 0
        try:
            result = self.expr()
            if self.index == len(self.expression):
                return "Notación postfija: " + result
            else:
                raise SyntaxError("Expresión mal formada: caracteres no
procesados al final")
        except SyntaxError as e:
            return str(e)

    def expr(self):
        """
        Maneja las expresiones con + y -
        expr -> term { ('+' | '-') term }
        """
        left = self.term()

        while self.index < len(self.expression) and
self.expression[self.index] in ('+', '-'):
            op = self.expression[self.index]
            self.index += 1
            right = self.term()
            # Construimos la notación postfija: primero operandos, luego
operador
            left = left + " " + right + " " + op
```

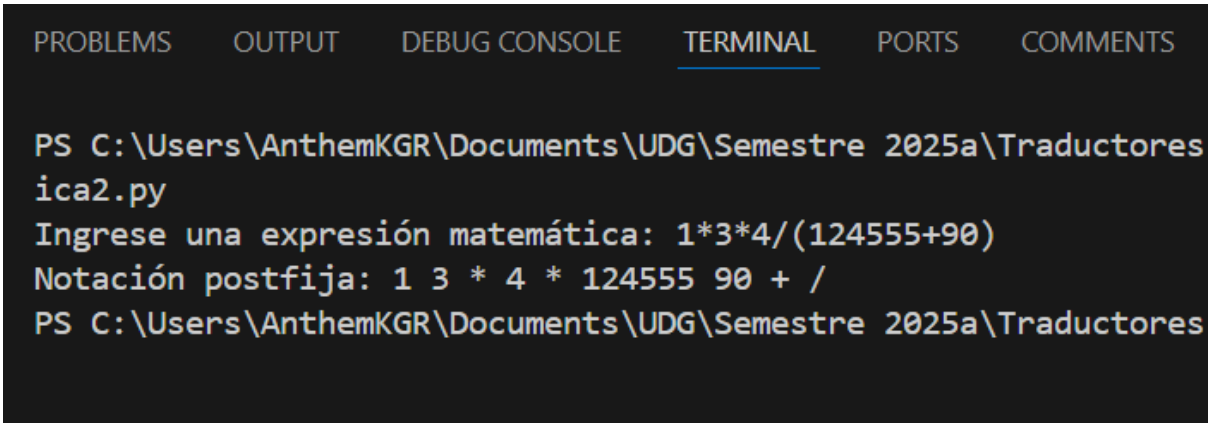
```
        return left

    def term(self):
        """
        Maneja las expresiones con * y /
        term -> factor { ('*' | '/') factor }
        """
        left = self.factor()

        while self.index < len(self.expression) and
self.expression[self.index] in ('*', '/'):
            op = self.expression[self.index]
            self.index += 1
            right = self.factor()
            # Construimos la notación postfija: primero operandos, luego
operador
            left = left + " " + right + " " + op

        return left

    def factor(self):
        """
        factor -> '(' expr ')' | number
        """
        if self.index < len(self.expression) and self.expression[self.index]
== '(':
            self.index += 1
            result = self.expr()
            if self.index < len(self.expression) and
self.expression[self.index] == ')':
                self.index += 1
                return result
            else:
                raise SyntaxError("Falta cerrar paréntesis")
        elif self.index < len(self.expression) and
self.expression[self.index].isdigit():
            return self.number()
        else:
            raise SyntaxError("Número o paréntesis esperado")

    def number(self):
        """
        Procesa números enteros
        number -> digit { digit }
        """
        num = ''
```

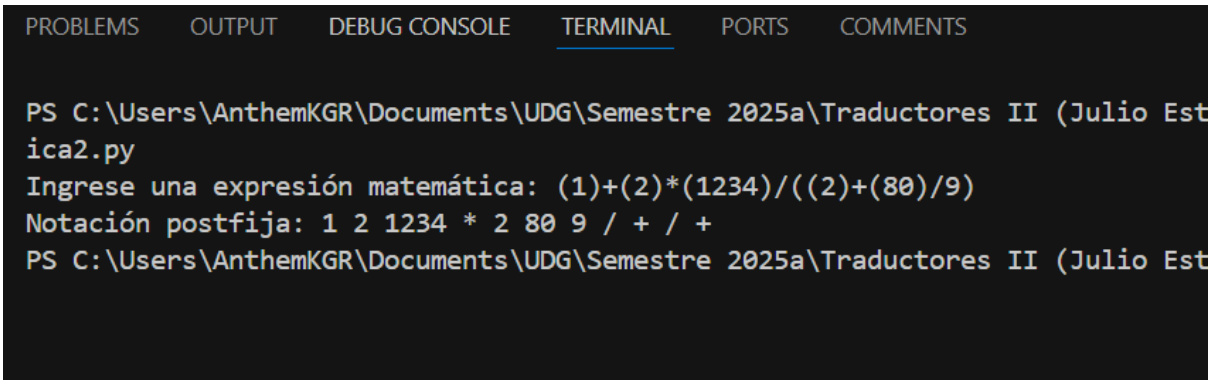
```
        while self.index < len(self.expression) and
self.expression[self.index].isdigit():
            num += self.expression[self.index]
            self.index += 1
        return num

# Ejemplo de uso
if __name__ == "__main__":
    expression = input("Ingrese una expresión matemática: ")
    parser = Parser(expression)
    result = parser.parse()
    print(result)
"""Código creado por Diego Andres Hernandez"""
```

Capturas de pantalla:









Conclusión:

DIEGO ANDRES HERNANDEZ RODRIGUEZ

Con esta actividad pude entender como es el funcionamiento de un analizador sintáctico y cómo se organiza la información cuando el compilador o intérprete tiene que hacer operaciones aritméticas si está trabajando con enteros. Esto debido a que la forma tradicional con la que nosotros trabajamos con operaciones y jerarquía de operaciones no es la misma con la que trabaja la computadora. De ahí nace la importancia de los analizadores sintácticos y otros analizadores que ya hemos visto.

Juan Antonio Perez Juarez

La verdad que aquí el viejon del DIego se la rifó con el Código, ya me lo explicó y la verdad que no es tan complicado, obviamente yo no lo hubiera podido hacer, sobre todo por que abstraer este tipo de problemas se me complica de sobremanera, por lo que yo ayudé poco la verdad.

Pero entiendo de manera general cómo es que funciona, y creo que con un mes de tiempo podría replicarlo por mi mismo.

Andre Maximiliano Calderon

Bueno, compadre, este código es un parser que agarra una expresión matemática y la convierte a notación postfija, o sea, acomoda los números y operadores para que se puedan evaluar más fácil después. Funciona siguiendo reglas básicas de matemáticas: primero multiplica y divide, luego suma y resta, y respeta los paréntesis pa' que no haya confusiones.

Lo bueno es que es ordenado y sigue las reglas de prioridad correctamente. Lo malo es que si la expresión está mal escrita, te manda un error, pero sin decirte exactamente dónde te equivocaste. En general, si metes una cuenta bien escrita, te la va a procesar bien, pero si te avientas un desorden, te va a regañar.

Bibliografía

colaboradores de Wikipedia. (2024, 4 junio). Analizador sintáctico. Wikipedia, la Enciclopedia Libre. https://es.wikipedia.org/wiki/Analizador_sint%C3%A1ctico