Proyecto: Math Expressions

Ariana Guadalupe Rosales Villalobos A01644773

Demmí Elizabeth Zepeda Rubio A01709619

Esteban Camilo Muñoz Rosero A01644609

Escuela de Ingeniería y Ciencias, Tecnológico de Monterrey

Implementación de Métodos Computacionales (Gpo. 603)

Profesor Obed Nehemías Muñoz Reynoso

28 de mayo de 2025

Descripción del Proyecto

Este proyecto consiste en el desarrollo de un analizador de expresiones matemáticas utilizando un autómata de pila, aplicando lo aprendido de los autómatas dentro de la teoría de la computación. El objetivo principal es validar si las expresiones matemáticas escritas por el usuario están correctamente construidas según las reglas de convención PEMDAS.

El compilador busca reconocer expresiones que incluyen operaciones básicas como suma (+), resta (−), multiplicación (*), división (/) y potenciación (**), así como números reales tanto positivos y negativos. Adicionalmente a esto, se busca que maneje el balanceo entre paréntesis (), y corchetes [], para agrupar operaciones y al mismo tiempo respetar la jerarquía de cada uno al momento de evaluar las expresiones.

El programa realizará un análisis sintáctico de cada expresión, comprobando que los operadores y operandos estén en el orden correcto, que los paréntesis se encuentren balanceados, y que no existan errores como operadores sin operandos o agrupaciones incompletas.

El analizador recibe como entrada un archivo de texto, *expressions.txt*, con múltiples expresiones matemáticas y genera como salida otro archivo, *results.txt*, donde indica si cada expresión es válida o inválida.

Definición del Autómata de Pila (PDA)

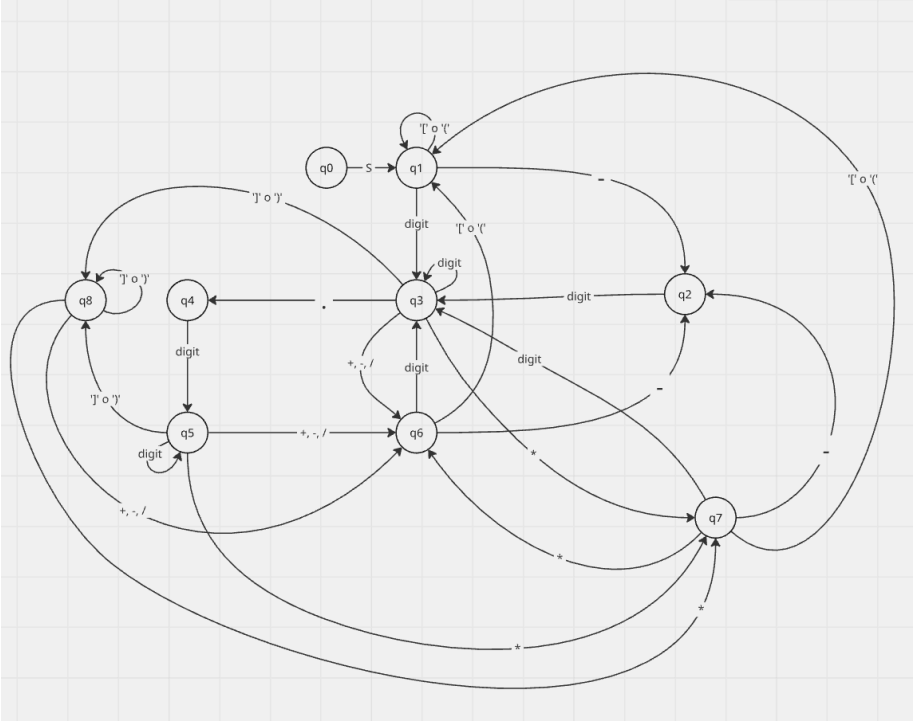

Figura 1. Diseño de la definición del autómata.

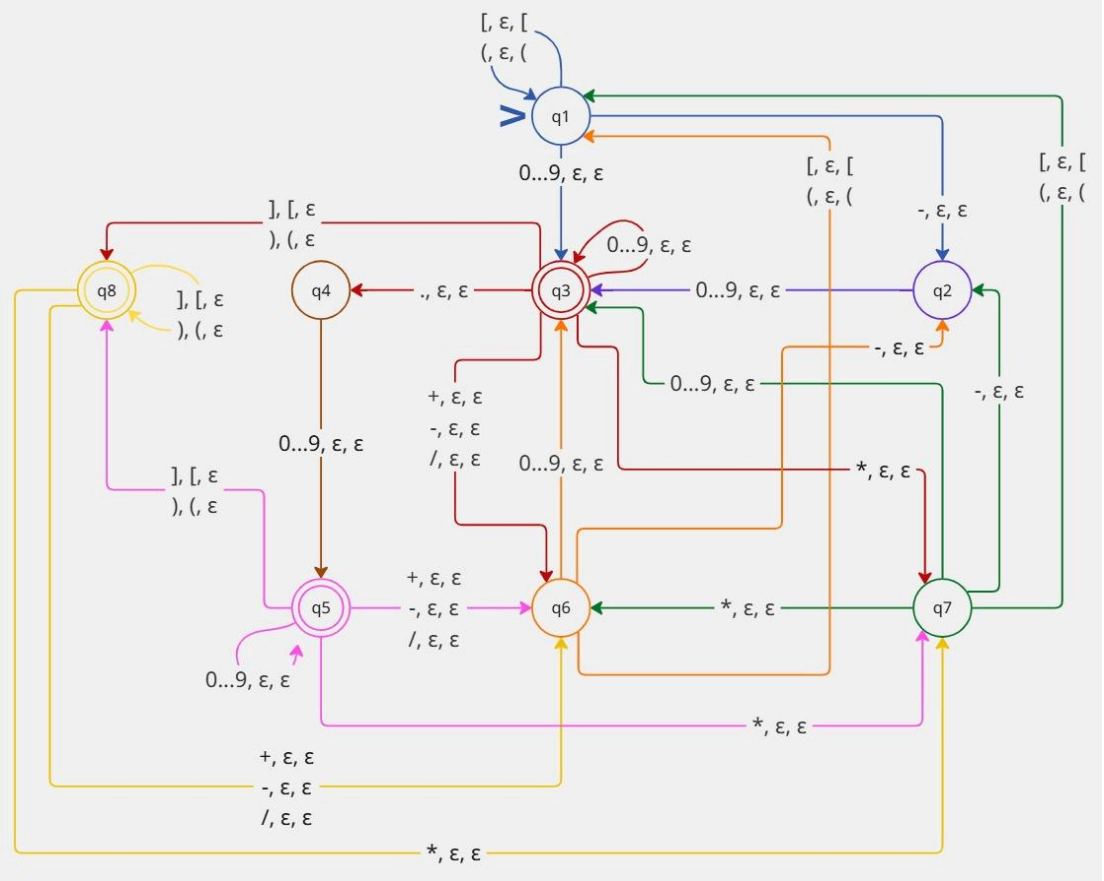

Figura 2. Definición formal del autómata de pila.

Ejemplo del Funcionamiento de PDA

Se define entonces un ejemplo del funcionamiento donde:

- El estado $q1$ es el inicio.
- La pila inicial solo afecta a corchetes y paréntesis.
- Al inicio la pila está vacía.
- Se desapila cuando los corchetes y paréntesis se cierran.

Pasamos a demostrar con un ejemplo las transiciones de una cadena que recibe: $(4 ** (3223 + 4))$ predispuesta como válida.

Estado	**Falta leer**	**Pila**	**Siguiente Estado**
q1	4**(3223+4))	(	q3
q3	**(3223+4))	(	q7
q7	*(3223+4))	(	q6
q6	(3223+4))	(	q1
q1	3223+4))	((	q3
q3	223+4))	((	q3
q3	23+4))	((	q3
q3	3+4))	((	q3
q3	+4))	((	q6
q6	4))	((	q3
q3	))	(	q8
q8	)	ε	q8
q8	ε	ε	fin

Proceso del Diseño del Programa

El código en general implementa un validador de expresiones matemáticas a través de la función $validateExpression()$. En la función $main()$ se leen las expresiones matemáticas desde un archivo de entrada, verifica si cada una es válida con la anterior función mencionada, bajo las condiciones de que las expresiones cumplen con las reglas de sintaxis; el condicional $if\ isOperator()$ fue el más complejo de programar debido a que durante las pruebas de testeo con diferentes tipos de expresiones, se descubrió que había ciertas condiciones que se pasaban por alto, como la regla sobre no permitir que un operador no tuviera dígitos a sus lados.

Se implementaron tokens después para poder mantener registro de caracteres previos al actual, para cerciorarse de que todas las reglas se cumplieran.

```
Go
            // Validar operadores
            if isOperator(ch) {
                  // Checar operador potencia **
                  if ch == '*' && i+1 < length && expr[i+1] == '*' {
                        // Verificar que no esté al final
                        if i+2 >= length {
                              return false
                        }
                        next := rune(expr[i+2])
                        if isOperator(next) || next == ')' || next == ']'
{
                              return false
                        }
                        // Asegura que antes haya número o cierre de
paréntesis o corchete
                        if prevToken != "number" && prevToken != ")" &&
prevToken != "]" {
                              return false
                        }
                        prevToken = "operator"
                        i += 2
                        continue
                  }
```

Ejemplos de Input y Output

Para el archivo de $expressions.txt$, se usó:

```
Unset
5*7
5 * 5
[(5 + 5)/(6-4)]
(4*(3))
(33+/4)
(23*6)*(6)/66
(4**(3223+4))
3+3
4%4*(4*5)
(23-2)/(5**6)+32
32***4
121.2*675
323/0
4.5.4+7
454-22(efdf)
4e4x4
2312+121+33.22(7*6)
1@3
55&^45
4334x22
(4-4*(7-5))
(452342.33*6)
[[43*(6+6)*(4+5)-7]]
{[(4_4)]*[4-6]}
{[(4**4)]*[4-6]}
{[(4)4)]*[4-6]}
([(4_4)]*[4-6])
[(4_4)]*[4-6]
[(7*6)-(8+8)]-[(5-4)-(4*66)]
[(7*6)-(8+8)]-[(5-4)**(4**66)]
[(7*6)-(8+8)]/[(5-4)-(4*66)]
[(7*6)-(8_8)]*[(5-4)**(4*66)]
[(7*6)-(8()8)]-[(5-4)-(4*66)]
[(7/6)**(8+8)]*8[(5-4)&&(4*66)]
(23*6)*(6)/66*(444+3)*[(3-3)-(3-3)*(8/8)]
(23*6)*(6)/66*[(444+3)]*[(3-3)-((3-3)*(8/8)]
(23*6)*(6)/66*(444+3)*[(3-8)*(3-3)*(8/8)]
(23*6)*(6)/66*(444+3)*[(3-3)-(3-3)*(8/8)]
(23*6)*(6)/66*(444+3*[(3-3)-([3-3)*(8/8)]
(23*6)(6)/66**(444+3)*[(3-3)-(3-3)*(8/8)]
(6**6 / 2)
(6*6 / )
(6*6 / 3)*(4)
([5*6])
```

Y el resultado en $results.txt$, fue el siguiente:

```
Unset
Expression 1: 5*7                                        -  Valid
Expression 2: 5 * 5                                      -  Valid
Expression 3: [(5 + 5)/(6-4)]                            -  Valid
Expression 4: (4*(3))                                    -  Valid
Expression 5: (33+/4)                                    -  Invalid
Expression 6: (23*6)*(6)/66                              -  Valid
Expression 7: (4**(3223+4))                              -  Valid
Expression 8: 3+3                                        -  Valid
Expression 9: 4%4*(4*5)                                  -  Invalid
Expression 10: (23-2)/(5**6)+32                           -  Valid
Expression 11: 32***4                                     -  Invalid
Expression 12: 121.2*675                                  -  Valid
Expression 13: 323/0                                      -  Valid
Expression 14: 4.5.4+7                                    -  Invalid
Expression 15: 454-22(efdf)                               -  Invalid
Expression 16: 4e4x4                                      -  Invalid
Expression 17: 2312+121+33.22(7*6)                        -  Invalid
Expression 18: 1@3                                        -  Invalid
Expression 19: 55&^45                                     -  Invalid
Expression 20: 4334x22                                    -  Invalid
Expression 21: (4-4*(7-5))                                -  Valid
Expression 22: (452342.33*6)                              -  Valid
Expression 23: [[43*(6+6)*(4+5)-7]]                       -  Valid
Expression 24: {[(4_4)]*[4-6]}                            -  Invalid
Expression 25: {[(4**4)]*[4-6]}                           -  Invalid
Expression 26: {[(4)4)]*[4-6]}                            -  Invalid
Expression 27: ([(4_4)]*[4-6])                            -  Invalid
Expression 28: [(4_4)]*[4-6]                              -  Invalid
Expression 29: [(7*6)-(8+8)]-[(5-4)-(4*66)]               -  Valid
Expression 30: [(7*6)-(8+8)]-[(5-4)**(4**66)]             -  Valid
Expression 31: [(7*6)-(8+8)]/[(5-4)-(4*66)]               -  Valid
Expression 32: [(7*6)-(8_8)]*[(5-4)**(4*66)]              -  Invalid
Expression 33: [(7*6)-(8()8)]-[(5-4)-(4*66)]              -  Invalid
Expression 34: [(7/6)**(8+8)]*8[(5-4)&&(4*66)]            -  Invalid
Expression 35: (23*6)*(6)/66*(444+3)*[(3-3)-(3-3)*(8/8)] -  Valid
Expression 36: (23*6)*(6)/66*[(444+3)]*[(3-3)-((3-3)*(8/8)] -  Invalid
Expression 37: (23*6)*(6)/66*(444+3)*[(3-8)*(3-3)*(8/8)] -  Valid
Expression 38: (23*6)*(6)/66*(444+3)*[(3-3)-(3-3)*(8/8)] -  Valid
Expression 39: (23*6)*(6)/66*(444+3*[(3-3)-([3-3)*(8/8)] -  Invalid
Expression 40: (23*6)(6)/66**(444+3)*[(3-3)-(3-3)*(8/8)] -  Invalid
Expression 41: (6**6 / 2)                                 -  Valid
Expression 42: (6*6 / )                                   -  Invalid
Expression 43: (6*6 / 3)*(4)                              -  Valid
Expression 44: ([5*6])                                    -  Invalid
```

Conclusiones

Esteban Muñoz

Este proyecto fue una muy buena oportunidad para aplicar los temas que vimos en teoría de la computación de una forma práctica. Implementar un analizador de expresiones me ayudó a entender mucho mejor cómo funcionan las gramáticas libres de contexto y autómatas de pila. Me gustó ver cómo conceptos como jerarquía de operadores, paréntesis balanceados o reglas de producción pueden reflejarse en el diseño de un parser. Aunque al principio fue un poco complejo estructurar bien la lógica y validar todas las combinaciones posibles, al final fue muy satisfactorio ver que el programa detectaba con precisión si una expresión era válida o no. También fue un buen ejercicio para mejorar mi lógica de programación y organizar mejor mi código. En general, siento que este proyecto me dio una base más sólida para entender cómo se construyen lenguajes y analizadores desde cero.

Ariana

Con este proyecto pudimos afinar nuestro entendimiento acerca de los autómatas de pila como una forma de representar lenguajes libres de contexto. Iniciando por definir el diagrama de estados y las transiciones para pasar por cada estado tratando de evitar redundancia o bucles haciendo pruebas múltiples. Nos topamos con complicaciones para adaptar este diagrama y sus las transiciones del diagrama dentro del código, siendo que go es un lenguaje con el que llevo poco tiempo familiarizada, fue preciso recurrir a actividades pasadas para darnos una idea más clara de cómo construir un compilador funcional. Aprendimos a manejar funciones con el uso de condicionales y booleanos con condiciones cuidadosamente específicas para poder lograr un resultado óptimo para validar la sintaxis de las cadenas. Creo que podemos profundizar aún más en la construcción de compiladores para

proyectos pasados pero considero que se consolidan de mejor manera las bases de lo que vimos durante clase.

Demmi

Durante la realización del proyecto en Golang, descubrí que disfruté bastante la elaboración del autómata en el que finalmente nos basamos para realizar la programación del compilador; me encontré muy interesada en definir el proceso de los estados en el diagrama, y además pude repasar con mayor atención lo que en clase ya habíamos hecho. Creo que lo más difícil en el proyecto fue pasar la lógica del autómata a código en Golang, y fue lo más tardado, puesto que había ciertas pruebas que por más que queríamos corregir, el compilador validaba expresiones no ideales, pero se pudo arreglar. El uso de un stack en el código me recordó a mis trabajos en estructura de datos y algoritmos, y no había trabajado con tokens tal cual de la manera en que se manejó en el código de este proyecto, entonces pude aprender algo nuevo. En general, fue un proyecto interesante y desafiante, y me dio una perspectiva más amplia respecto a la elaboración de compiladores, en donde me cuestioné qué procesos más complejos se involucran en la realización de aparatos o software.