Universidad Rafael Landívar Facultad de ingeniería en informática y sistemas Lenguajes Formales y Autómatas
Sección: 2 Ing. Vivian Damaris Campos González

Diseño de Expresiones Regulares

Brandon Enrique Salazar Guevara - 1111322
José Daniel Paz Ortega - 1200022

Guatemala, 22 de octubre del 2025

Estructura General del Escáner

Inicialización: En el constructor __init__, se inicializan dos listas:

- self.tokens: Almacena los tokens válidos encontrados (objetos de la clase Token).
- self.errores: Almacena reportes de errores (objetos de la clase ReporteError para lexemas inválidos).

Entrada y Salida: El método analizar_texto recibe una cadena de texto (text: str) y devuelve una tupla con las listas de tokens y errores.

Procesamiento por Líneas: Divide el texto en líneas usando text.split('\n') para manejar filas y columnas correctamente (útil para reportar posiciones en errores).

- fila comienza en 1 y se incrementa por cada línea.
- Para cada línea, col comienza en 1 y i es el índice del carácter actual.

Bucle Principal: Es un bucle while i < len(line) que avanza carácter por carácter, saltando espacios en blanco (char.isspace()).

- Para cada carácter no-espacio, se determina si inicia un token conocido.
- Si no coincide con ningún patrón, se registra como error.

Enfoque de Reconocimiento: Es un **escáner determinista** que usa condicionales para clasificar el carácter inicial y luego recolecta el lexema completo en un buffer. Esto simula un autómata finito determinista (DFA) manualmente, donde cada rama del if representa una transición basada en el tipo de carácter.

Tipos de Tokens Reconocidos

Los tokens se definen en el enum TipoToken (originalmente TokenType):

- *MENOR* ("<"): Para etiquetas de apertura.
- *MAYOR* (">"): Para cierre de etiquetas.
- *BARRA* ("/"): Para etiquetas de cierre (e.g., </Operacion>).
- *IGUAL* ("="): Para atributos (e.g., <Operacion=SUMA>).
- *IDENTIFICADOR* ("Identifier"): Palabras como "OPERACION", "NUMERO", "SUMA", etc. (normalizadas a mayúsculas).
- *NUMERO* ("Number"): Números enteros o decimales (e.g., "123", "45.67", ".89").
- *ERROR* ("Error"): No es un token válido, pero se usa internamente para reportes.

El escáner no reconoce strings literales o otros tipos complejos, ya que el lenguaje parece limitado a etiquetas XML y valores numéricos.

Flujo Detallado de Reconocimiento de Tokens

El bucle procesa cada línea carácter por carácter. Para cada char = line[i] no-espacio:

- Se guarda la posición inicial (fila_inicio, col_inicio) para reportar en el token o error.
- Luego, se ramifica según el tipo de char:

a. Tokens Especiales:

- Condición: if char in "<>/=".
- Diseño: Estos son tokens de un solo carácter, reconocidos directamente sin buffer.
- Expresión Regular Implícita: Cada uno equivale a una regex simple como ^<$, ^>$, etc. (pero implementado manualmente).
- Proceso:
  - Mapea el carácter a su tipo usando un diccionario: e.g., "<" → TipoToken.MENOR.
  - Crea un Token con el tipo, lexema (el carácter mismo), y posición.
  - Avanza i += 1 y col += 1.
- Ejemplo: Si encuentra "<", agrega Token(MENOR, "<", fila, col).

b. Identificadores:

- Condición: if char.isalpha() (debe empezar con una letra).
- Diseño: Usa un buffer para recolectar el lexema completo.
- Expresión Regular Implícita: Equivale a ^[a-zA-Z][a-zA-Z0-9]*$ (letra seguida de cero o más alfanuméricos).
  - No usa regex explícita; en su lugar, recolecta manualmente.
- Proceso:
  - Inicializa buffer = char.
  - Avanza i += 1, col += 1.

        - Bucle while i < len(line) and line[i].isalnum(): Agrega line[i] al buffer y avanza.
        - Normaliza a mayúsculas: buffer.upper().
        - Crea Token(IDENTIFICADOR, buffer.upper(), fila_inicio, col_inicio).
    - Ignora case (todo a upper), lo que simplifica el parsing posterior. No permite guiones u otros caracteres, solo alfanuméricos.

c. Números:

- Condición: if char.isdigit() or (char == '.' and i+1 < len(line) and line[i+1].isdigit()).
    - Permite empezar con dígito o con punto decimal si le sigue un dígito (para números como ".5").
- Diseño: Similar a identificadores, usa un buffer y un flag decimal para rastrear si ya se vio un punto.
- Expresión Regular Explícita: Aquí sí usa regex para **validación** después de recolectar el buffer.
    - Valida con re.match(r'^\d+(\.\d+)?$', buffer) para números como "123" o "123.45".
    - O re.match(r'^\.\d+$', buffer) para números como ".45".
    - Estas regex aseguran:
        - ^\d+(\.\d+)?$: Comienza con uno o más dígitos, opcionalmente seguido de "." y uno o más dígitos. No permite "." solo o al final.
        - ^\.\d+$: Comienza con ".", seguido de uno o más dígitos (para fracciones puras).
    - Si no coincide, registra como error.
- Proceso Detallado:
    - Inicializa buffer = char, decimal = (char == '.').
    - Avanza i += 1, col += 1.
    - Bucle while i < len(line):
        - Si line[i].isdigit(): Agrega al buffer, avanza.

        - Si line[i] == '.' and not decimal: Agrega ".", setea decimal = True, avanza.
        - Else: Sale del bucle (fin del número).
    - Valida con las regex:
        - Si válido: Crea Token(NUMERO, buffer, fila_inicio, col_inicio).
        - Si inválido: Agrega ReporteError(buffer, fila_inicio, col_inicio).
- No permite notación científica (e.g., "1e3"), negativos (no hay condición para "-"), o comas. El flag decimal previene múltiples puntos (e.g., "1.2.3" se recolecta como "1.2." y falla en regex).

d. Errores (Cualquier Otro Carácter):

- Condición: Ninguna de las anteriores.
- Diseño: Cualquier carácter no reconocido se trata como error individual.
- Proceso: Agrega ReporteError(char, fila_inicio, col_inicio), avanza i += 1, col += 1.
- Expresión Regular Implícita: Cualquier cosa que no coincida con los patrones anteriores.
- Notas: Para secuencias inválidas en números/identificadores, se recolecta el buffer y se valida; si falla, todo el buffer es un error

Manejo de Posiciones y Errores

Posiciones: Cada token/error guarda *fila* y *col* del inicio del lexema, lo que facilita depuración y reportes.

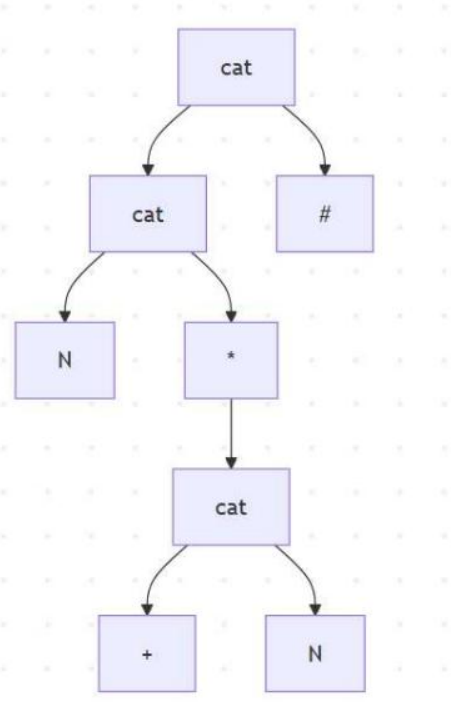
Espacios: Saltados completamente (continue), no generan tokens.

Fin de Línea/Archivo: El bucle por líneas asegura que no se pierdan posiciones.

Errores Acumulados: No detiene el escaneo; continúa para reportar todos.

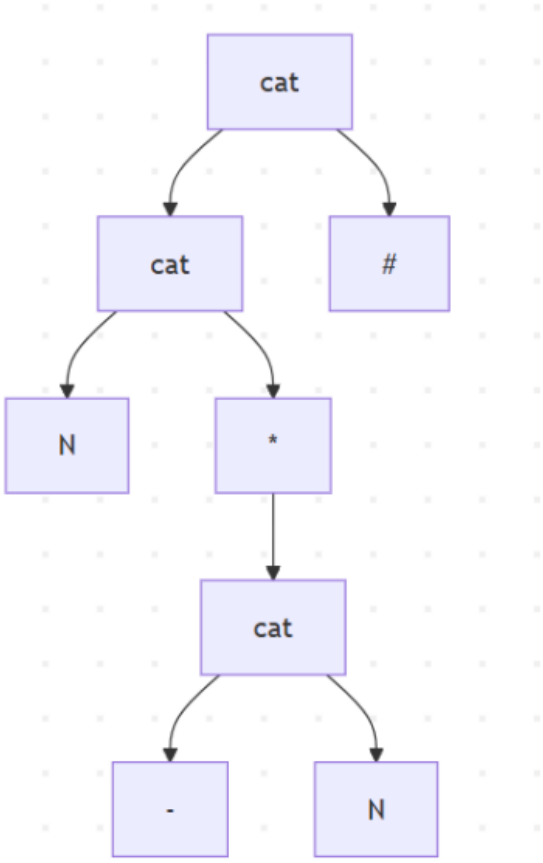
1. SUMA (ER: N (+ N)* #)

- **Razones del diseño**:
    - En el código, SUMA es n-aria: evaluate hace sum(children_vals), sumando todos los hijos (números u operaciones anidadas).
    - El parser permite múltiples <Numero> o <Operacion> como hijos, lo que se traduce en una secuencia de N separados por + (abstracto).
    - Debe empezar con al menos un N (no suma vacía, como en el código que requiere hijos válidos).
    - El * permite cero o más repeticiones de (+ N), cubriendo unary (solo N), binary (N + N), etc.
    - '+' es el op simbólico (del método to_str en Operation).
    - Augmento con # para el método del árbol (posición final de aceptación).
- **Finalidad**: Se analizo evaluate para SUMA (suma todos), y el loop en parse_operacion que recolecta hijos hasta el cierre. Esto es equivalente a una ER para listas separadas por op.
- **Tokens aceptados**: Secuencias como N, N+N, N+N+N (donde N es un número válido).



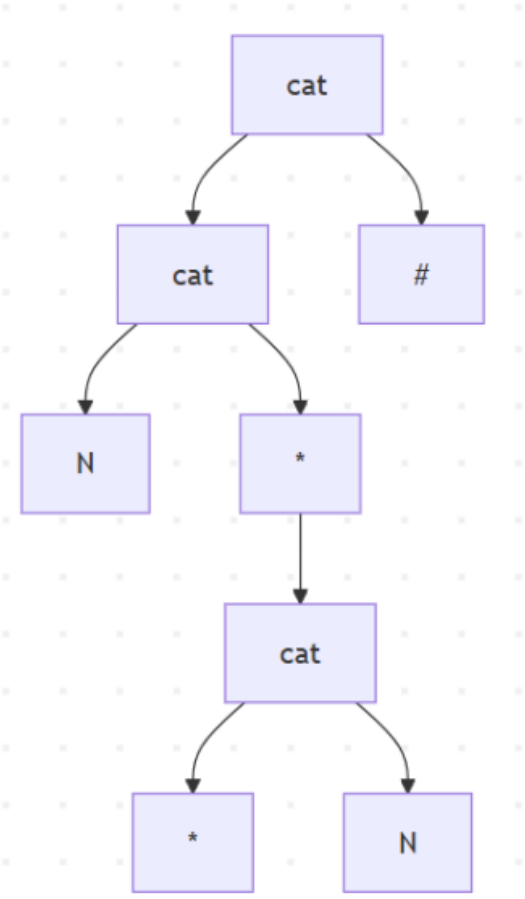
2. RESTA (ER: N (- N)* #)

- **Razones del diseño**:
    - Similar a SUMA: evaluate hace res = children_vals[0]; then res -= v for each subsequent v (asociatividad izquierda).
    - Permite múltiples operandos: unary (solo N, que es N sin resta), binary (N - N), n-ary (N - N - N = (N - N) - N).
    - Empieza con N (primer operando), luego cero o más (- N).
    - '-' del to_str.
- **Finalidad**: De evaluate para RESTA. El parser recolecta múltiples hijos, modelado como lista separada por -.
- **Tokens aceptados**: N, N-N, N-N-N.



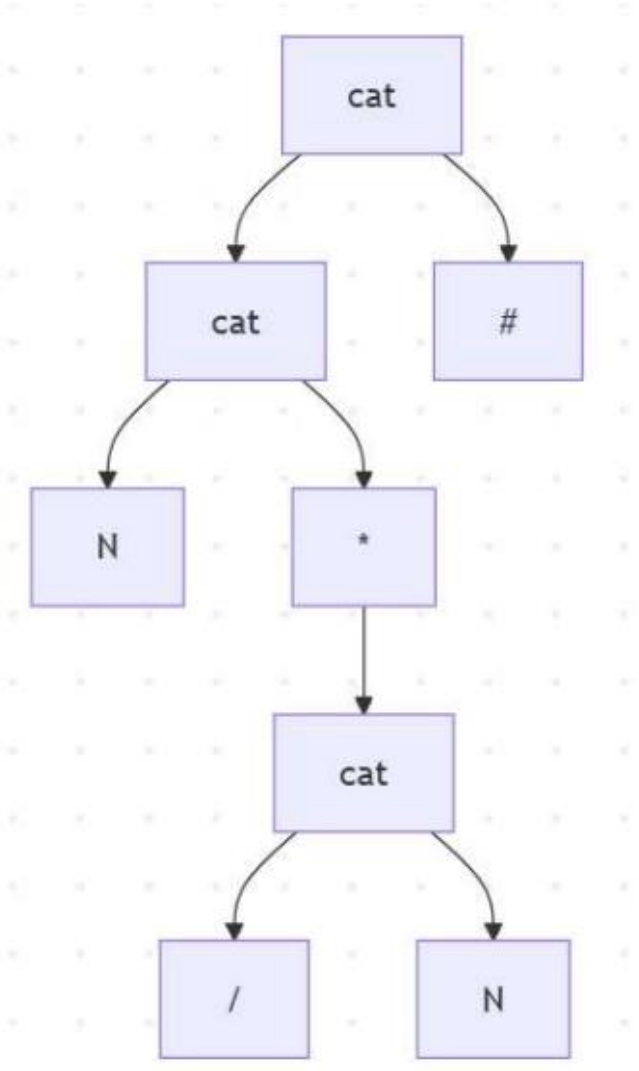
3. MULTIPLICACION (ER: N (* N)* #)

- **Razones del diseño**:
  - evaluate: res = 1; then res *= v for each v (multiplica todos, iniciando en 1 si vacío, pero código requiere hijos).
  - N-aria: unary (N), binary (N * N), etc.
  - Empieza con N, cero o más (* N).
  - '*' del to_str.
- **Finalidad**: De evaluate. Parser permite múltiples hijos, como lista separada por *.
- **Tokens aceptados**: N, N*N, NN*N.



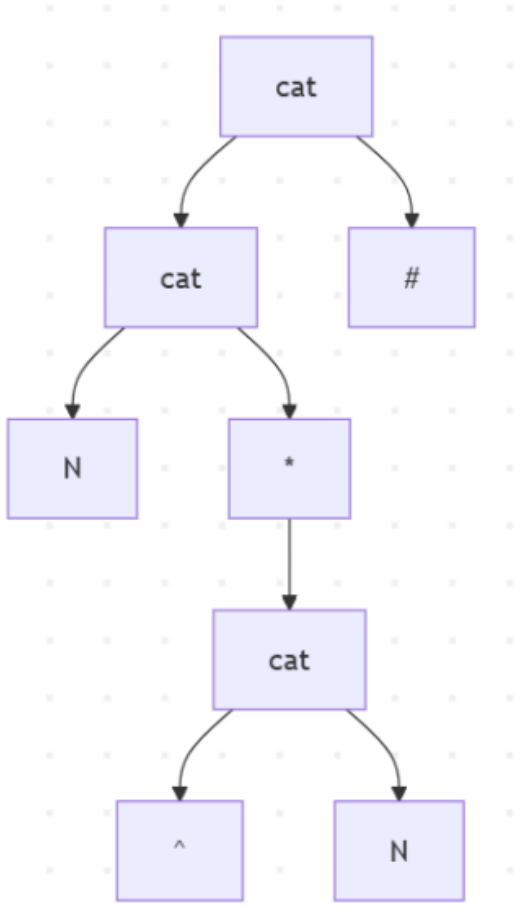
4. DIVISION (ER: N (/ N)* #)

- **Razones del diseño**:
  - evaluate: res = children_vals[0]; then res /= v for each subsequent.
  - Asociatividad izquierda: N / N / N = (N / N) / N.
  - Empieza con N, cero o más (/ N).
  - '/' del to_str.
- **Finalidad**: Similar a RESTA. Múltiples hijos en parser.
- **Tokens aceptados**: N, N/N, N/N/N.



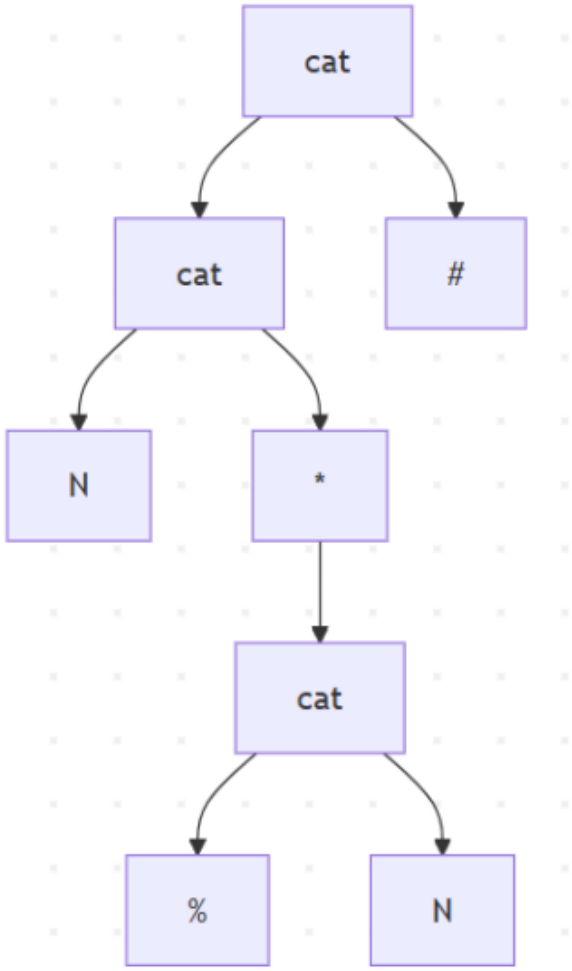
5. POTENCIA (ER: N (^ N)* #)

- **Razones del diseño**:
  - evaluate: res = children_vals[0]; then res **= v for each subsequent (asociatividad derecha: N ^ N ^ N = N ^ (N ^ N)).
  - Aún n-aria: unary (N), binary (N ^ N), etc.
  - La ER modela la secuencia lineal (no la asociatividad, que es en evaluación), así que misma estructura: N seguido de cero o más (^ N).
  - '^' del to_str.
- **Finalidad**: De evaluate (potencia acumulativa derecha). Parser permite múltiples hijos, como lista separada por ^. La asociatividad no afecta la ER (solo la secuencia de tokens).
- **Tokens aceptados**: N, N^N, N^N^N.



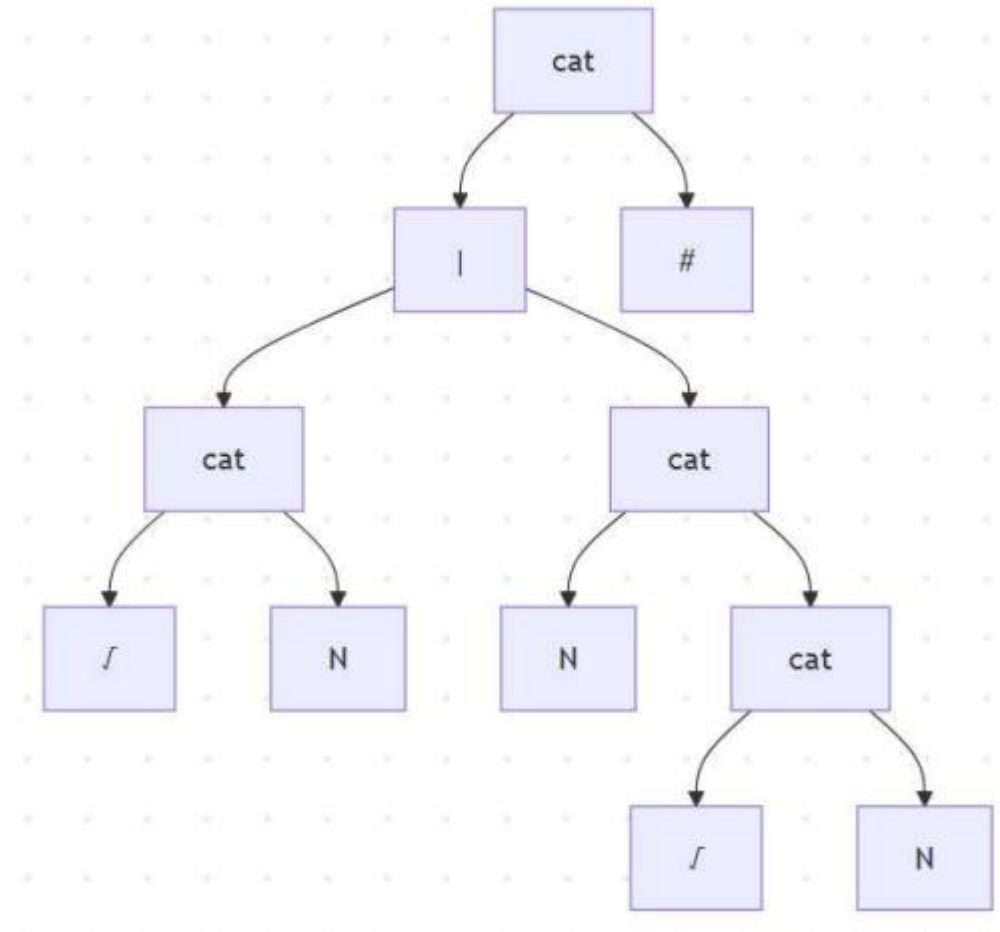
6. MOD (ER: N (% N)* #)

- **Razones del diseño**:
    - evaluate: res = children_vals[0]; then res %= v for each subsequent (asociatividad izquierda).
    - N-aria: unary (N), binary (N % N), etc.
    - Empieza con N, cero o más (% N).
    - '%' del to_str.
- **Finalidad**: Similar a RESTA/DIVISION (acumula módulo desde el primero). Múltiples hijos en parser.
- **Tokens aceptados**: N, N%N, N%N%N.



7. RAIZ (ER: (√ N) | (N √ N) #)

- **Razones del diseño**:
    - evaluate: Si 1 hijo: N ** 0.5 (raíz cuadrada unary). Si 2 hijos: base ** (1 / index) (binary: index √ base).
    - Código permite 1 o 2 hijos (no más, basado en if len(children_vals) == 1 else).
    - ER con alternación | : unary (√ N) o binary (N √ N).
    - '√' del to_str (con opcional [index]).
- **Finalidad**: De evaluate (condicional unary/binary). Parser recolecta hijos, pero eval limita a 1-2. No * para repeticiones, solo | para casos.
- **Tokens aceptados**: √N (unary), N√N (binary). No más (e.g., no N√N√N).



INVERSO (ER: INV N #)

- **Razones del diseño**:
    - evaluate: 1 / children_vals[0] (siempre unary, un solo hijo).
    - No permite múltiples (código asume len==1).
    - ER simple: INV seguido de N (INV abstracto para "1/").
    - '1/' del to_str.
- **Finalidad**: De evaluate (inverso de un solo valor). Parser recolecta hijos, pero eval usa solo el primero. ER estricta unary.
- **Tokens aceptados**: INV N (e.g., 1/N). No múltiples.