Implementación de un Analizador Léxico Manual para Expresiones Aritméticas

Tu Nombre Completo
Carrera o Curso: Lenguajes y Autómatas
Matrícula: [Tu Matrícula]

1 de diciembre de 2025

Resumen

Este documento describe el diseño e implementación de un Analizador Léxico (Scanner) para un subconjunto de expresiones aritméticas. El analizador se construyó manualmente, simulando un Autómata Finito Determinista (AFD) en el lenguaje C, sin recurrir a bibliotecas de expresiones regulares. Se detalla la estructura del AFD, la tabla de transiciones y la solución léxica implementada para resolver la ambigüedad del símbolo de resta/negación.

1. Introducción

El Análisis Léxico es la primera fase del proceso de compilación, responsable de transformar una secuencia de caracteres de entrada en una secuencia de unidades atómicas llamadas *tokens*. Este proyecto se enfoca en la construcción manual de un **Autómata Finito Determinista (AFD)** capaz de reconocer los tokens de operadores, paréntesis y números (enteros y decimales) dentro de una expresión aritmética. El principal desafío abordado es la contextualización del símbolo menos ($-$) para distinguirlo entre el operador binario de resta y el operador unario de negación.

2. Diseño del Autómata Finito Determinista (AFD)

2.1. Diagrama de Transiciones

El AFD se diseñó para cubrir todos los requisitos léxicos especificados.

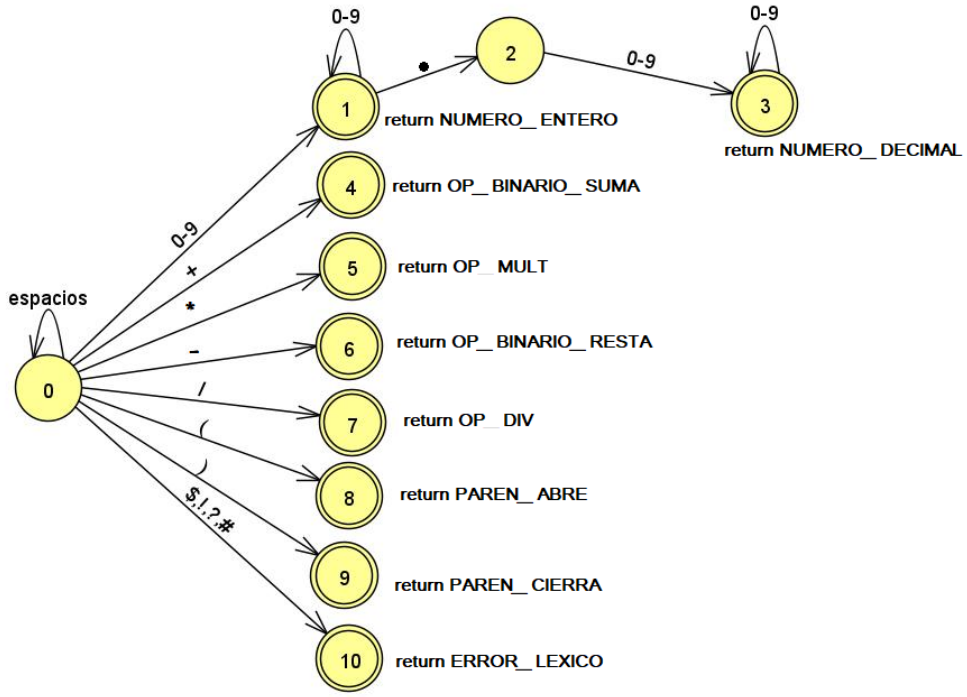

Figura 1: Diagrama de Estados del Autómata Finito Determinista.

2.2. Tabla de Transiciones

Esta tabla es el corazón lógico de la implementación y refleja el comportamiento de los estados definidos.

Q	0-9	+	-	*	/	(	)	.	$,!,?,# y otros	¿Estado de aceptación?
0	**1**	**4**	**6**	**5**	**7**	**8**	**9**	-1	-1	-1
1	-1	-1	-1	-1	-1	-1	-1	**2**	-1	return NUMERO_NTERO
2	**3**	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	return NUMERO_DECIMAL
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	return OP_BINARIO_SUMA
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	return OP_MULT
6	-1	-1	-1	-1	-1	-1	-1	-1	-1	return OP_BINARIO_RESTA
7	-1	-1	-1	-1	-1	-1	-1	-1	-1	return OP_DIV
8	-1	-1	-1	-1	-1	-1	-1	-1	-1	return PAREN_ABRE
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	return PAREN_CIERRA
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	return ERROR_LEXICO

Figura 2: Tabla de Transiciones de un Analizador Léxico.

3. Implementación del Analizador Léxico

El analizador se implementó en `C` utilizando estructuras `enum` para representar los tipos de tokens (`TipoToken`) y los estados del AFD (`EstadoAFD`). La función principal `getNextToken` itera sobre la cadena de entrada y utiliza una estructura `switch` anidada para simular las transiciones de estados del AFD.

3.1. Definición de Tokens y Estados

Se definieron las siguientes estructuras en el código para tipificar la salida y la lógica interna:

```
typedef enum {
    TOKEN_INICIO,
    NUMERO_ENTERO,
    NUMERO_DECIMAL,
    OP_MULT,
    OP_DIV,
    OP_BINARIO_SUMA,
    OP_BINARIO_RESTA,
    OP_UNARIO_NEG,
    PAREN_ABRE,
    PAREN_CIERRA,
    ERROR_LEXICO,
    FIN_CADENA
} TipoToken;

typedef struct {
    TipoToken tipo;
    char lexema[100];
} Token;

typedef enum {
    ESTADO_INICIAL,
    ESTADO_LEYENDO_ENTERO,
    ESTADO_ESPERA_DECIMAL,
    ESTADO_LEYENDO_DECIMAL,
    ESTADO_LEYENDO_OP
} EstadoAFD;
```

Listing 1: Definiciones de Tipos de Tokens y Estados del AFD

3.1.1. Análisis Detallado de la Función `getNextToken` con Extractos de Código

La función `getNextToken` es el corazón del analizador léxico, extrayendo el siguiente token válido de la expresión. A continuación, se detalla su mecanismo paso a paso, mostrando los fragmentos de código C que implementan cada lógica:

1. **Gestión de Estados y Posición:**

   Inicializa las variables de estado y posición, e ignora los espacios en blanco (`isspace(c)`) antes de iniciar la búsqueda del token, actualizando el punto de inicio del lexema.

```
EstadoAFD estado_actual = ESTADO_INICIAL;
int inicio_lexema = *indice;
// ...
// Ignorar espacios en ESTADO_INICIAL
if (estado_actual == ESTADO_INICIAL && isspace(c)) {
    (*indice)++;
    inicio_lexema = *indice;
    continue;
}
```

2. **Reconocimiento de Tokens de Un Carácter:**

   Desde `ESTADO_INICIAL`, reconoce operadores binarios (`+`, `*`, `/`) y paréntesis (`(` y `)`). Estos tokens se identifican inmediatamente, se avanza el índice y se salta a la finalización (`goto token_encontrado`).

```
case ESTADO_INICIAL:
    // ...
    if (c == '+') {
        nuevo_token.tipo = OP_BINARIO_SUMA;
        (*indice)++;
        goto token_encontrado;
    } else if (c == '*' || c == '/') {
        nuevo_token.tipo = (c == '*') ? OP_MULT : OP_DIV;
        (*indice)++;
        goto token_encontrado;
    } else if (c == '(') {
        nuevo_token.tipo = PAREN_ABRE;
        (*indice)++;
        goto token_encontrado;
    } else if (c == ')') {
        nuevo_token.tipo = PAREN_CIERRA;
        (*indice)++;
        goto token_encontrado;
    }
    // ...
```

3. **Reconocimiento de Números (AFD):**

   El Autómata Finito Determinista (AFD) maneja la lectura de números, permitiendo el reconocimiento de enteros y decimales. Si se abandona la lectura de dígitos, se finaliza el token, clasificándolo como `NUMERO_ENTERO` o `NUMERO_DECIMAL`.

```
case ESTADO_INICIAL:
    if (isdigit(c)) {
        estado_actual = ESTADO_LEYENDO_ENTERO;
        (*indice)++;
    } // ...
    break;

case ESTADO_LEYENDO_ENTERO:
    if (isdigit(c)) {
        (*indice)++;
    } else if (c == '.') {
        estado_actual = ESTADO_ESPERA_DECIMAL;
        (*indice)++;
    } else {
        nuevo_token.tipo = NUMERO_ENTERO;
        goto token_encontrado;
    }
    break;

case ESTADO_ESPERA_DECIMAL:
    if (isdigit(c)) {
        estado_actual = ESTADO_LEYENDO_DECIMAL;
        (*indice)++;
    } else {
        nuevo_token.tipo = NUMERO_DECIMAL;
        goto token_encontrado;
    }
    break;

case ESTADO_LEYENDO_DECIMAL:
    if (isdigit(c)) {
```

```
32          (*indice)++;
33      } else {
34          nuevo_token.tipo = NUMERO_DECIMAL;
35          goto token_encontrado;
36      }
37      break;
```

4. **Finalización (`token_encontrado`):**

   Una vez que se ha identificado el tipo de token y se ha determinado su límite, el código utiliza las posiciones `inicio_lexema` e `*indice` para extraer la subcadena (el **lexema**) y almacenarla en la estructura `Token` antes de salir.

```
1  token_encontrado:
2      {
3          int longitud_lexema = *indice - inicio_lexema;
4          if (longitud_lexema > 0 && nuevo_token.tipo != FIN_CADENA) {
5              // Copiar el segmento de la expresion al campo lexema
6              strncpy(nuevo_token.lexema, expresion + inicio_lexema,
                   longitud_lexema);
7              nuevo_token.lexema[longitud_lexema] = '\0';
8          } else if (nuevo_token.tipo == FIN_CADENA) {
9              strcpy(nuevo_token.lexema, "EOF");
10         } else {
11             // Para tokens de un solo caracter
12             nuevo_token.lexema[0] = expresion[inicio_lexema];
13             nuevo_token.lexema[1] = '\0';
14         }
15     }
16     return nuevo_token;
```

3.2. Manejo de la Ambigüedad del Signo Menos (−/Negación)

El desafío clave del proyecto fue resolver la ambigüedad entre el operador binario de **Resta** (`OP_BINARIO_RESTA`) y el operador **Unario Negativo** (`OP_UNARIO_NEG`). La solución implementada se basa en el **contexto** del token previamente reconocido (`token_anterior`).

- **Negación Unaria (`OP_UNARIO_NEG`):** El símbolo − se clasifica como negación si el token anterior es un operador binario (+, ∗, /), un paréntesis de apertura ((), o si es el inicio de la expresión.
- **Resta Binaria (`OP_BINARIO_RESTA`):** El símbolo − se clasifica como resta si el token anterior fue un número (`NUMERO_ENTERO`, `NUMERO_DECIMAL`) o un paréntesis de cierre ()).

El siguiente extracto de código muestra la lógica de decisión en el estado inicial para el símbolo menos (extraído de `projectv5.c`):

```
1  } else if (c == '-') {
2      if (token_anterior == TOKEN_INICIO ||
3          token_anterior == OP_BINARIO_SUMA ||
4          token_anterior == OP_BINARIO_RESTA ||
5          token_anterior == OP_MULT ||
6          token_anterior == OP_DIV ||
7          token_anterior == PAREN_ABRE) {
8
9          nuevo_token.tipo = OP_UNARIO_NEG;
10     } else {
11         nuevo_token.tipo = OP_BINARIO_RESTA;
```

```
    }
    (*indice)++;
    goto token_encontrado;
}
```

Listing 2: Fragmento de código C para la ambigüedad del signo menos

4. Resultados y Pruebas

Se realizaron pruebas con diferentes cadenas de expresiones aritméticas para verificar la correcta clasificación de los tokens, especialmente en casos de ambigüedad y manejo de errores léxicos.

- **Entrada de Prueba:** Se utilizó la expresión que incluye el desafío de ambigüedad.

```
2.5 + -3 * (-10.5) / (5 - 3)
5$ + 1 + 4.1.2
```

Listing 3: Entrada de Prueba (input.txt)

- **Salida del Analizador:** La salida demuestra la correcta distinción entre `OP_UNARIO_NEG` y `OP_BINARIO_RESTA`, así como la detección de errores.

```
Expresion: 2.5 + -3 * (-10.5) / (5 - 3)

---------------  -------
TOKEN            LEXEMA
---------------  -------
NUMERO_DECIMAL   2.5
OP_BINARIO_SUMA  +
OP_UNARIO_NEG    -
NUMERO_ENTERO    3
OP_MULT          *
PAREN_ABRE       (
OP_UNARIO_NEG    -
NUMERO_DECIMAL   10.5
PAREN_CIERRA     )
OP_DIV           /
PAREN_ABRE       (
NUMERO_ENTERO    5
OP_BINARIO_RESTA -
NUMERO_ENTERO    3
PAREN_CIERRA     )
FIN_CADENA       EOF

Expresion: 5$ + 1 + 4.1.2
--------------- -------
TOKEN       LEXEMA
--------------- -------
NUMERO_ENTERO   5
ERROR_LEXICO    $
OP_BINARIO_SUMA +
NUMERO_ENTERO   1
OP_BINARIO_SUMA +
NUMERO_DECIMAL  4.1
ERROR_LEXICO    .
NUMERO_ENTERO   2
FIN_CADENA      EOF
```

Listing 4: Salida Generada (output.txt)

5. Conclusiones

La implementación del Analizador Léxico a través de la simulación manual de un AFD demostró ser efectiva para reconocer y clasificar correctamente los tokens de expresiones aritméticas. El uso de la variable de contexto (`token_anterior`) fue crucial para resolver la ambigüedad del signo menos, cumpliendo con los objetivos del proyecto.

Referencias

[1] Varad Ingale, Abhishek Yeole, Kuldeep Vayadande, Sahil Zawar, Vivek Verma, and Zoya Jamadar. Lexical analyzer using DFA. *International Journal of Advance Research, Ideas and Innovations in Technology (IJARIIT)*, 8:161–164, 2022.

[2] Hassan k. Mohamed, Fathia A. A. Albadri, and Raja A. Mohamed. Lexical analysis implementation by using deterministic finite automata (dfa). *(Al-Manara Scientific Journal)*, 3:32–39, November 2021.