UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Lenguajes formales y autómatas

Proyecto:

Reconocedor sintáctico de operaciones con números reales

Profesora: Lucila Patricia Arellano Mendoza

Grupo: 4

Alumnos:

- Alvarez Saldaña Ariana Lizeth
- Rosas Gomez Fernando

Fecha de entrega: 25 de Abril del 2023

PROYECTO

Descripción:

Realizar un reconocedor sintáctico de operaciones con números reales, debe reconocer expresiones que involucren números con o sin signo, punto decimal obligatorio y por lo menos debe de reconocer una operación entre números reales (pueden ser más de una). Dicho reconocedor debe ser programado en lenguaje C. Se deberán de incluir ejemplos de ejecución.

Ejemplos:

-5.7 / +14.6 = 234.9393 - - 4.291+29.18*0.234 =

0.6 * - .54 = -.5629 + - 0.4938*5.17-54.28+-0.33/-549 =

Para probarlo se introducirá una cadena que involucra las características mencionadas, el reconocedor nos dirá si la expresión es correcta, es decir, pertenece al lenguaje descrito. Se deberá de entregar un trabajo escrito con todo el análisis realizado y el código utilizado en lenguaje C, además de un archivo .zip que contendrá el trabajo escrito y el archivo ejecutable .exe para poder probarlo.

I. Expresión Regular

Objetivo: Se identifica, analiza y expresa con la notación matemática correspondiente la expresión regular que corresponde a la solución al problema planteado, toma en consideración todos los elementos necesarios para formar, incluyendo la gramática utilizada.

Desarrollo:

Se crea un lenguaje regular para los números reales:

$$d = \{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9\}$$

Estos números requieren de un valor positivo o un valor negativo, por lo que se crea un lenguaje regular que contenga estos símbolos:

$$s = \{+,\ -,\ \}$$

La concatenación de estos va con un punto decimal obligatorio, por lo que se crea un lenguaje regular para este:

$$p = \{.\}$$

Además, se crea un lenguaje regular para las operaciones que pueden realizar los numeros:

$$o = \{+,\ -,\ *,\ /\}$$

Por último, se crea un lenguaje regular con el propósito de ser el símbolo de igual, es decir, “=”:

$$i = \{=\}$$

Para llevar a cabo la generación de expresiones como se solicitan en la descripción, se inicia con la concatenación del valor o signo que pueden adoptar los números, junto con los números reales. Para que esta expresión acepte cualquier número que se introduce, se usa la cerradura positiva.

$$w = sd^{+} \quad (1)$$

Este término debe siempre llevar un punto decimal, el cual concatena con la expresión (1):

$$w = (sd^{+})p \quad (2)$$

Después del punto decimal, deben de ir más números, por lo que usando a (2), se concatena con los números reales que cuentan con la operación de cerradura positiva:

$$w = (sd^{+})p(d^{+}) \quad (3)$$

Con la expresión (3), podemos crear expresiones como

- -231.5543
- +327785.1276

Estas expresiones deben de ir con una operación de la siguiente forma:

$$w = [(sd^{+})p(d^{+})]o \quad (4)$$

Según las indicaciones de la descripciones, se debe de realizar por lo menos una operación entre los números reales, siendo que la expresión que va después de la operación debe de ser expresada exactamente a lo obtenido en (3):

$$w = [(sd^{+})p(d^{+})]o[(sd^{+})p(d^{+})] \quad (5)$$

Pero para que se puedan realizar mas de una operación, la expresión (5) se cambia por la siguiente forma:

$$w = ([(sd^{+}) p (d^{+})] o^{*}) \quad (6)$$

En la expresión 6, el conjunto de "*o*" se le asigna una cerradura estrella para que las operaciones se realicen ya sea ninguna vez, es decir, tener solo hasta la expresión (3), o más de una vez.
Ahora bien,se puede observar que la expresión (5) se repite 2 veces la expresión (3), indicando la operación entre 2 números reales, pero para que se realicen más operaciones entre más de 2 números reales, se usa la cerradura positiva sobre toda expresión (6):

$$w = ([(sd^{+}) p (d^{+})] o^{*})^{+} \quad (7)$$

Por último al final de cada expresión regular debe de tener el símbolo de "=" , concatenando a toda la expresión (7) con el conjunto "*i*"

$$w = ([(sd^{+}) p (d^{+})] o^{*})^{+} i \quad (8)$$

Algunas cadenas que se pueden obtener son:

$$([(sd^{+}) p (d^{+})] o^{*})^{+} i = sdpdi,\ sdpddi,\ sdpdosdpdi,\ sdddpdddosddpdddi\ldots$$

También se puede describir como:

$$([(sd^{+}) p (d^{+})] o^{*})^{+} i = (sd^{+}\ p\ d^{+}\ o^{*})^{+} i$$

$$= (sd^{n+1}\ p\ d^{n+1}\ o^{n})^{n+1} i$$

II. Análisis y diseño

Objetivo

Realiza el análisis paso a paso hasta obtener el autómata finito determinístico mínimo correspondiente a la expresión regular.

Desarrollo

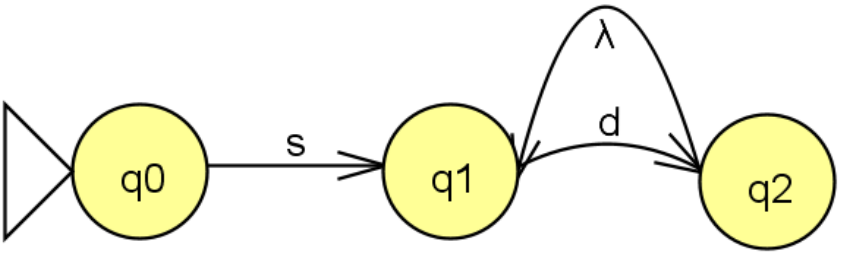
La expresión regular es:

$$(\ [(sd^{+}) \ p \ (d^{+}) \ ] \ o^{*} \ )^{+} \ i$$

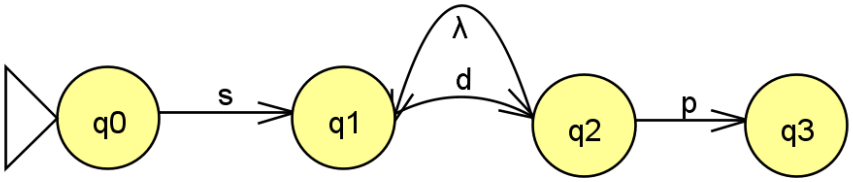
Esta expresión se representará con un autómata a partir del diagrama de Moore y tablas de transición, y se construye con la metodología de Thompson para llegar al autómata finito determinístico.

A continuación se muestra la construcción del autómata paso a paso. Esta primera representación es la de un autómata finito no determinístico (**AFND**), debido a las transiciones con **λ**:

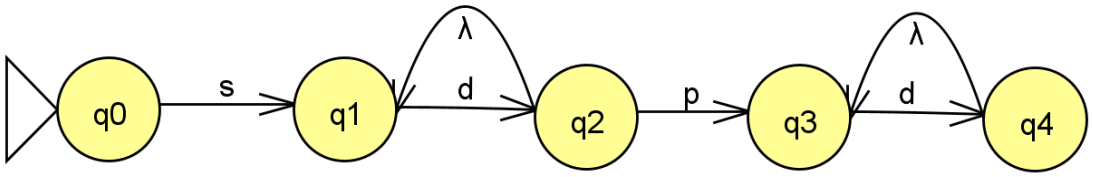
- (sd^{+})



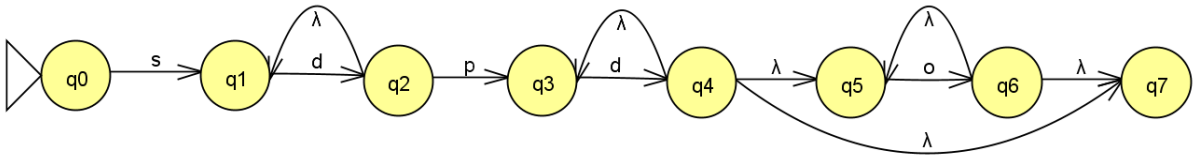
- $(sd^{+}) \ p$



- $(sd^{+}) \ p \ (d^{+})$

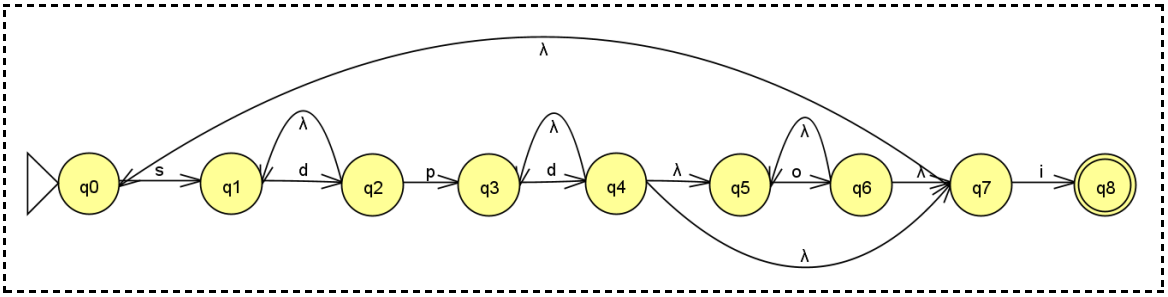


- $[(sd^{+})\ p\ (d^{+})\ ]\ o^{*}$



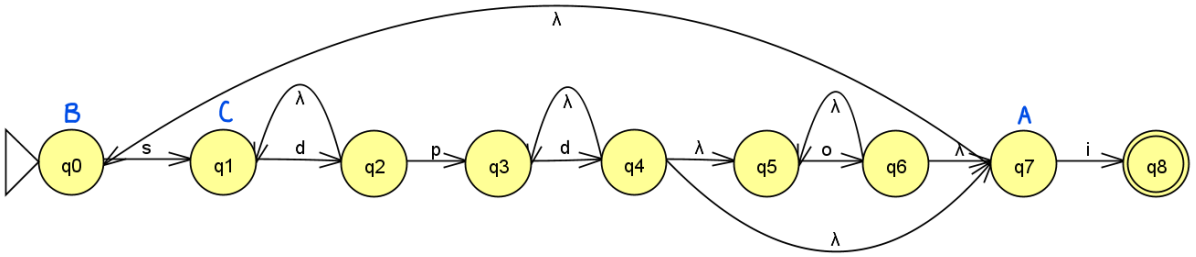
Obtención del AFND

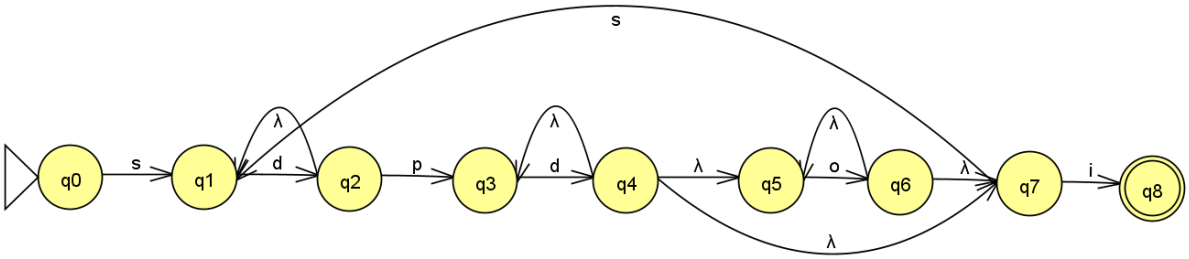
- $(\ [(sd^{+})\ p\ (d^{+})\ ]\ o^{*}\ )^{+}\ i$



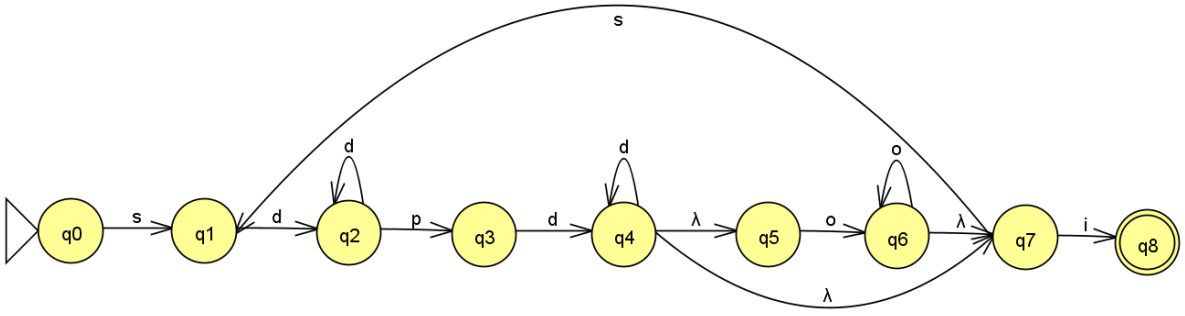
Ahora se lleva a cabo la eliminación de transiciones λ paso a paso:

1.

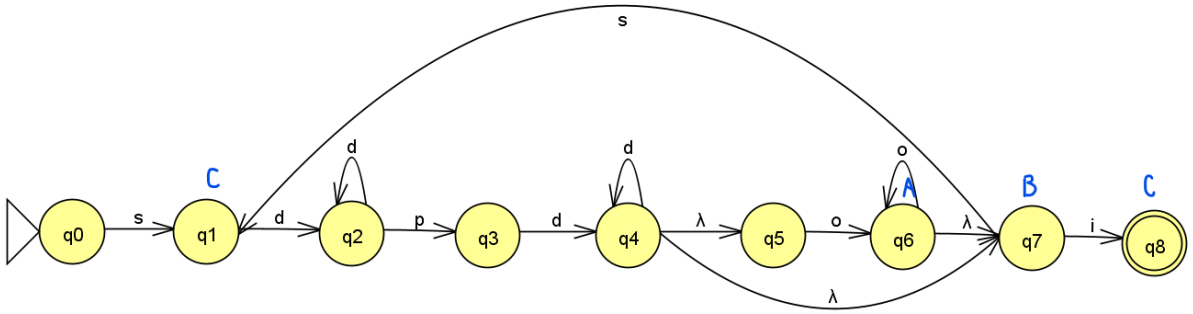


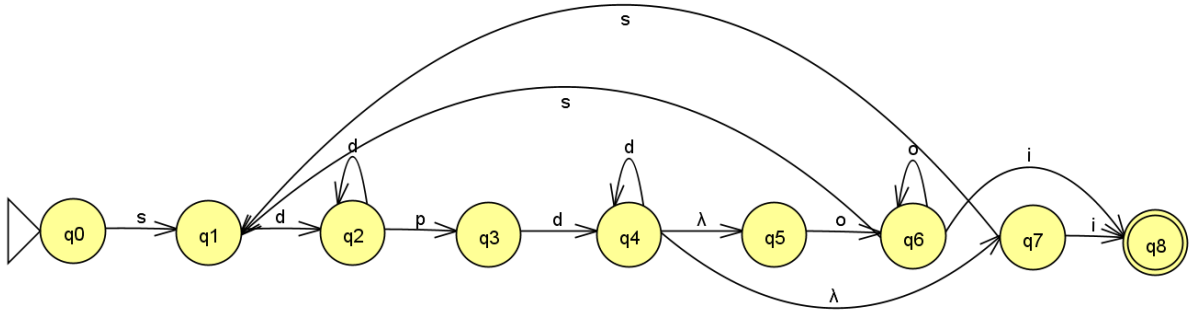


2.

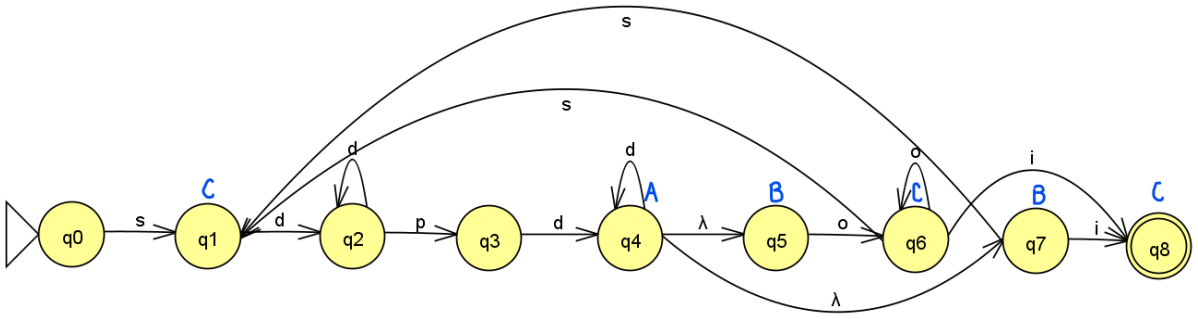


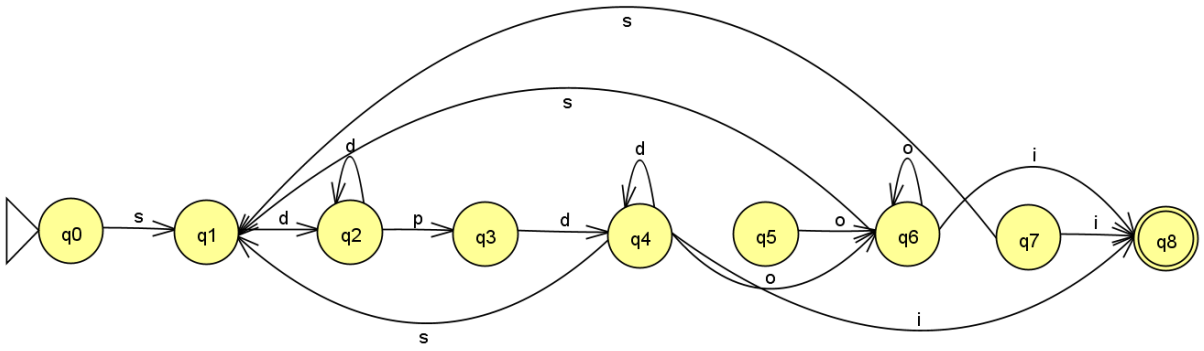
3.


q0
q1
q2
q3
q4
q5
q6
q7
q8
s
d
p
λ
o
i
C
A
B
C

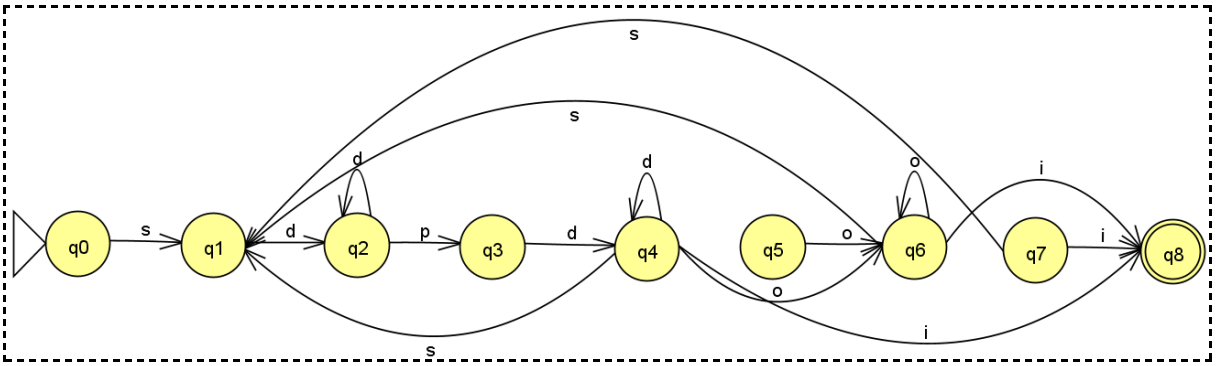

q0
q1
q2
q3
q4
q5
q6
q7
q8
s
d
p
λ
o
i

4.


q0
q1
q2
q3
q4
q5
q6
q7
q8
s
d
p
λ
o
i
C
A
B
C
B
C


q0
q1
q2
q3
q4
q5
q6
q7
q8
s
d
p
o
i

Obtención del AFND sin transiciones con λ.



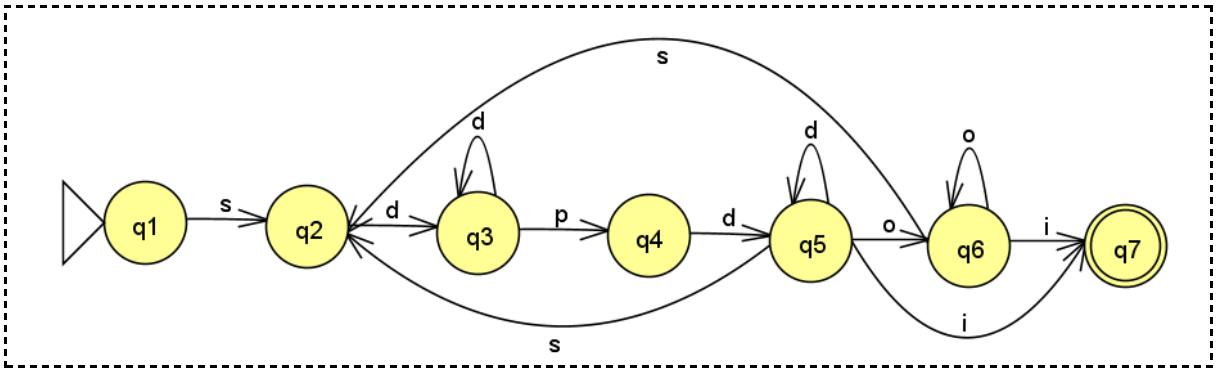
Para obtener el Autómata Finito determinístico (**AFD**) se realiza la tabla de transiciones los estados actuales:

R	Q	s	d	p	o	i	A/R
1	q0	q1	-	-	-	-	0
2	q1	-	q2	-	-	-	0
3	q2	-	q2	q3	-	-	0
4	q3	-	q4	-	-	-	0
5	q4	q1	q4	-	q6	q8	0
6	q6	q1	-	-	q6	q8	0
7	q8	-	-	-	-	-	1

Con la tabla anterior se aprecia que los estados q5 y q7 son estados extraños, es decir, son estados que producen pero ningún estado llega a este. Con esto en mente, se asignan nuevos estados dando lugar a un AFD y se crea un nuevo diagrama.

	Q	s	d	p	o	i	A/R
1	q1	q2	-	-	-	-	0
2	q2	-	q3	-	-	-	0
3	q3	-	q3	q4	-	-	0
4	q4	-	q5	-	-	-	0
5	q5	q2	q5	-	q6	q7	0
6	q6	q2	-	-	q6	q7	0
7	q7	-	-	-	-	-	1

Obtención de un AFD a partir de AFND



Se juntos aquellos estados que sean equivalentes, que son los estados que producen las mismas cadenas que otro, sin embargo, en este caso no hay dicha equivalencia, tal y como se muestra en la siguiente tabla:

	Q	s	d	p	o	i	A/R
1	q1	q2	-	-	-	-	0
2	q2	-	q3	-	-	-	0
3	q3	-	q3	q4	-	-	0
4	q4	-	q5	-	-	-	0
5	q5	q2	q5	-	q6	q7	0
6	q6	q2	-	-	q6	q7	0
7	q7	-	-	-	-	-	1

Al carecer de estados equivalentes, se avanza a la creación del AFD mínimo. Para obtener el AFD mínimo se separan los estados terminales de los no terminales, y se separan en 2 grupos diferentes.

Q	s	d	p	o	i	
q1	q2	-	-	-	-	A
q2	-	q3	-	-	-	A
q3	-	q3	q4	-	-	A
q4	-	q5	-	-	-	A
q5	q2	q5	-	q6	q7	A
q6	q2	-	-	q6	q7	A
q7	-	-	-	-	-	B

Se cambian los estados por su respectivo grupo y se crean los nuevos grupos.Este paso se repite hasta que exista un grupo diferente por cada uno de los estados.

Q	s	d	p	o	i	
q1	A	-	-	-	-	A
q2	-	A	-	-	-	B
q3	-	A	A	-	-	C
q4	-	A	-	-	-	B
q5	A	A	-	A	B	D
q6	A	-	-	A	B	E
q7	-	-	-	-	-	F

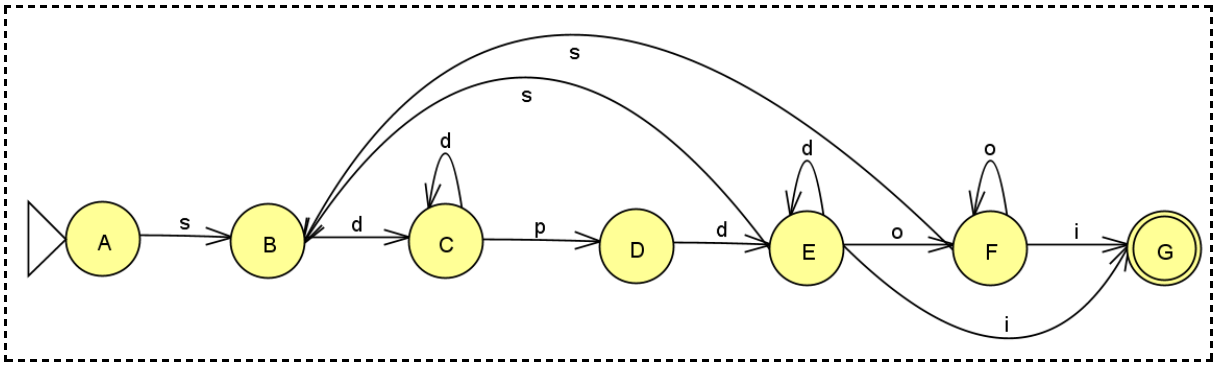
Q	s	d	p	o	i	
q1	B	-	-	-	-	A
q2	-	C	-	-	-	B
q3	-	C	B	-	-	C
q4	-	D	-	-	-	D
q5	B	D	-	E	F	E
q6	B	-	-	E	F	F
q7	-	-	-	-	-	G

La cantidad de grupos es igual a la cantidad de estados, por lo que si en la siguiente tabla esto se mantiene, se da por finalizado el análisis

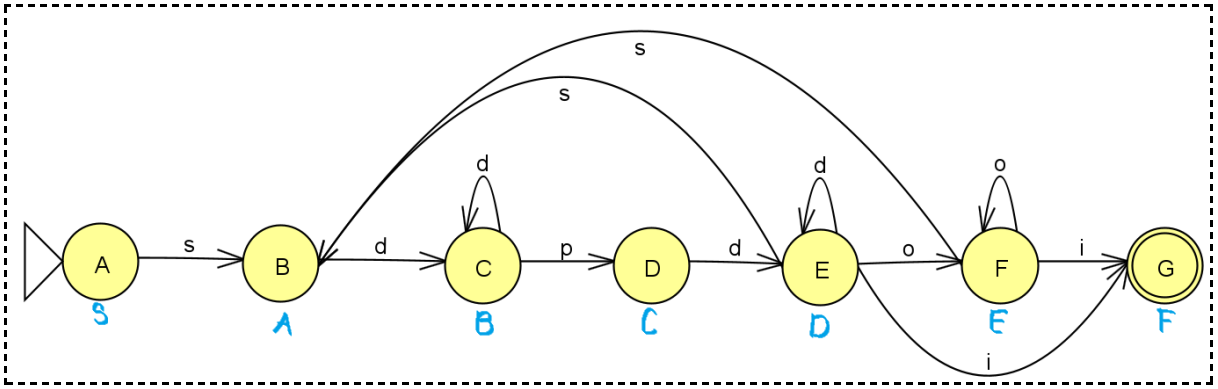
Q	s	d	p	o	i	
q1	B	-	-	-	-	A
q2	-	C	-	-	-	B
q3	-	C	D	-	-	C
q4	-	E	-	-	-	D
q5	B	E	-	F	G	E
q6	B	-	-	F	G	F
q7	-	-	-	-	-	G

Es en esta última tabla en la que se dibuja el ultimo diagrama del automata, siendo este un AFD Mínimo.

Obtención del AFD Mínimo



Obtención de una Gramática Regular a partir de un Autómata Finito



$$
\begin{aligned}
G = \{ & \\
& S \rightarrow sA \qquad D \rightarrow oE \\
& A \rightarrow dB \qquad D \rightarrow sA \\
& B \rightarrow dB \qquad E \rightarrow sA \\
& B \rightarrow pC \qquad E \rightarrow oE \\
& C \rightarrow dD \qquad E \rightarrow iF \\
& D \rightarrow dD \qquad F \rightarrow \xi \\
& D \rightarrow iF \qquad \}
\end{aligned}
$$

III. Funcionalidad

Objetivo

A partir del análisis obtenido desarrolla la aplicación correspondiente en el lenguaje C, que le permite comprobar el buen funcionamiento de la solución al problema planteado.

Desarrollo

Código completo

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int pos = 0;  // Posición actual en la cadena de entrada

// Prototipos de funciones
double parseExpr(char* input);
double parseTerm(char* input);
double parseFactor(char* input);
double parseNumber(char* input);

// Función principal
int main() {
    char input[100];
    printf("Introduce una operacion con numeros reales: ");
    fgets(input, sizeof(input), stdin);
    if (parseExpr(input) && input[pos] == '=') {
        printf("La operacion es correcta.\n");
    } else {
        printf("La operacion es incorrecta.\n");
```

```
        }
        return 0;
    }

    // Función para analizar una expresión
    double parseExpr(char* input) {
        double value = parseTerm(input);
        while (input[pos] == '+' || input[pos] ==
'-') {
            char op = input[pos++];
            double term = parseTerm(input);
            if (op == '+') {
                value += term;
            } else {
                value -= term;
            }
        }
        return value;
    }

    // Función para analizar un término
    double parseTerm(char* input) {
        double value = parseFactor(input);
        while (input[pos] == '*' || input[pos] ==
'/') {
            char op = input[pos++];
            double factor = parseFactor(input);
            if (op == '*') {
                value *= factor;
            } else {
```

```
                value /= factor;
            }
        }
        return value;
    }

    // Función para analizar un factor
    double parseFactor(char* input) {
        double value;
        if (input[pos] == '(') {
            pos++;
            value = parseExpr(input);
            pos++;  // Consumir el ')'
        } else {
            value = parseNumber(input);
        }
        return value;
    }

    // Función para analizar un número real
    double parseNumber(char* input) {
        double value = 0;
        int sign = 1;
        if (input[pos] == '-') {
            sign = -1;
            pos++;
        }
        while (isdigit(input[pos])) {
            value = value * 10 + (input[pos] -
'0');
```

```
            pos++;
        }
        if (input[pos] == '.') {
            pos++;
            double factor = 0.1;
            while (isdigit(input[pos])) {
                value += factor * (input[pos] -
'0');
                factor /= 10;
                pos++;
            }
        }
        return sign * value;
    }
```

Código detallado por partes

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int pos = 0; //Posición actual en la cadena de entrada
// Prototipos de funciones
double parseExpr(char* input);
double parseTerm(char* input);
double parseFactor(char* input);
double parseNumber(char* input);
```

Función: Se incluyen las bibliotecas necesarias y se declara una variable global de nombre “pos” con un valor de cero. En seguida se realizan las declaraciones de todos las funciones que se usan.

```c
// Función principal
int main() {
    char input[100];
    printf("Introduce una operacion con numeros reales: ");
    fgets(input, sizeof(input), stdin);
    if (parseExpr(input) && input[pos] == '=') {
        printf("La operacion es correcta.\n");
    } else {
        printf("La operacion es incorrecta.\n");
    }
    return 0;
}
```

Función: Se solicita al usuario que introduzca una operación de números reales. La función lee una cadena de entrada de nombre "input" usando la función fgets. En seguida se usa la estructura de if para validar que dicha cadena sea una operación de números reales. Es dentro de esta estructura en la que se usa la función parseExpr la cual analiza que la expresión sea correcta y que tiene el signo = al final. En caso de no cumplir esto, se indicará que la operación es incorrecta

```
// Función para analizar una expresión
double parseExpr(char* input) {
    double value = parseTerm(input);
    while (input[pos] == '+' || input[pos] == '-') {
        char op = input[pos++];
        double term = parseTerm(input);
        if (op == '+') {
            value += term;
        } else {
            value -= term;
        }
    }
    return value;
}
```

Función: La función parseExpr recibe una cadena de entrada y analiza una expresión matemática. Se declara una variable de nombre "value" la cual es igual a la función parseTerm para analizar cada término de la expresión. Con la estructura while, la función realiza una suma o resta de los términos según los operadores + y - que encuentre en la cadena

```
// Función para analizar un término
double parseTerm(char* input) {
    double value = parseFactor(input);
    while (input[pos] == '*' || input[pos] == '/') {
        char op = input[pos++];
        double factor = parseFactor(input);
        if (op == '*') {
            value *= factor;
        } else {
            value /= factor;
        }
    }
    return value;
}
```

```
// Función para analizar un término
double parseTerm(char* input) {
    double value = parseFactor(input);
    while (input[pos] == '*' || input[pos] == '/') {
        char op = input[pos++];
        double factor = parseFactor(input);
        if (op == '*') {
            value *= factor;
        } else {
            value /= factor;
        }
    }
    return value;
}
```

Función: La función recibe como argumento una cadena de entrada. Después se declara la variable "value" que es igual a la función parseFactor para analizar cada factor del término. Con la estructura while, mientras la cadena de entrada contenga el símbolo "*" o "/" la función realiza una multiplicación o división de los factores según los operadores.

```
// Función para analizar un factor
double parseFactor(char* input) {
    double value;
    if (input[pos] == '(') {
        pos++;
        value = parseExpr(input);
        pos++;  // Consumir el ')'
    } else {
        value = parseNumber(input);
    }
    return value;
}
```

Función: La función recibe como argumento una cadena. Al declararse la variable "value" si el factor comienza con un paréntesis, la función utiliza la función parseExpr para analizar la expresión que está entre paréntesis, de lo contrario la función utiliza la función parseNumber para analizar un número real.

```
// Función para analizar un número real
double parseNumber(char* input) {
    double value = 0;
    int sign = 1;
    if (input[pos] == '-') {
        sign = -1;
        pos++;
    }
    while (isdigit(input[pos])) {
        value = value * 10 + (input[pos] - '0');
        pos++;
    }
    if (input[pos] == '.') {
        pos++;
        double factor = 0.1;
        while (isdigit(input[pos])) {
            value += factor * (input[pos] - '0');
            factor /= 10;
            pos++;
        }
    }
    return sign * value;
}
```

Función: La función parseNumber recibe una cadena de entrada para analizarla como un número real. La función lee los dígitos del número y los va acumulando en la variable "value". La función toma en cuenta si el número de la entrada tiene signo y si cuenta con un punto decimal. Al final retorna el valor agregado a "value".

IV. Conclusiones

En conclusión, se creó un autómata no determinístico de operaciones con números reales que puede reconocer expresiones que involucren números con o sin signo, punto decimal obligatorio y al menos una operación entre números reales. Este autómata se convirtió en un autómata determinístico mínimo para obtener una forma más efectiva en la implementación de un analizador sintáctico
La conversión del autómata no determinístico a uno determinístico mínimo es un proceso que implica la eliminación de transiciones ε y la combinación de estados equivalentes(que no fue el caso). Este proceso permitió reducir el número de estados del autómata, lo que facilita su implementación.

Una vez convertido en un autómata determinístico mínimo, el analizador sintáctico podrá utilizarlo para validar correctamente la sintaxis de las expresiones y realizar las operaciones matemáticas correspondientes de manera precisa y eficiente. Además, la implementación de un autómata determinístico mínimo también facilita la detección de errores en la entrada del usuario y la emisión de mensajes de error claros y precisos.

Por otro lado, la creación del programa en C, requirió la definición de una gramática adecuada y la validación de la entrada del usuario para evitar errores y operaciones matemáticas no válidas.

Cuando el programa es creado, los usuarios pueden introducir expresiones matemáticas y el programa verificará si la sintaxis es correcta y si se pueden realizar las operaciones matemáticas correspondientes. En caso de que la sintaxis sea incorrecta, el programa emitirá un mensaje de error.

Por lo tanto, con este proyecto podemos concluir que la creación de un reconocedor sintactico puede ser implementado de una forma más fácil y eficiente si se inicia con la creación de un AFND y llevarlo a un AFD mínimo, para luego continuar con el proceso de obtener una Gramáticas de Contexto Libre