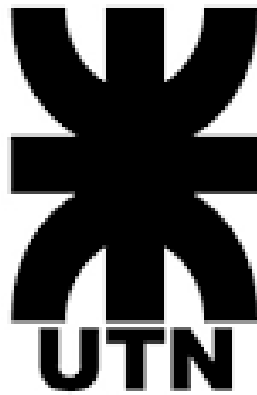


INFORME TP AUTÓMATAS

Materia: SSL

Explicación del pensamiento detrás del trabajo



Fecha de entrega: 1/10/2023

Link repositorio: <https://github.com/GFCACACE/ssl-tp-automatas-q25>

Integrantes Grupo 25:

-CACACE, Guillermo Federico

-CALÓ, Ignacio

-GOMEZ PEREYRA, Manuel Francisco

- MAJER, Cecilia Alejandra

-TROSSERO, Agustín Francisco

PUNTO 1

Antes de comenzar con el diseño de los autómatas, primero dedujimos la ERX testada en el html provisto en clase:

JUGUEMOS CON EXPRESIONES REGULARES

CADENA DE PRUEBA:

0x44A\$88\$077\$

Verificar expresion regular

RESULTADO:

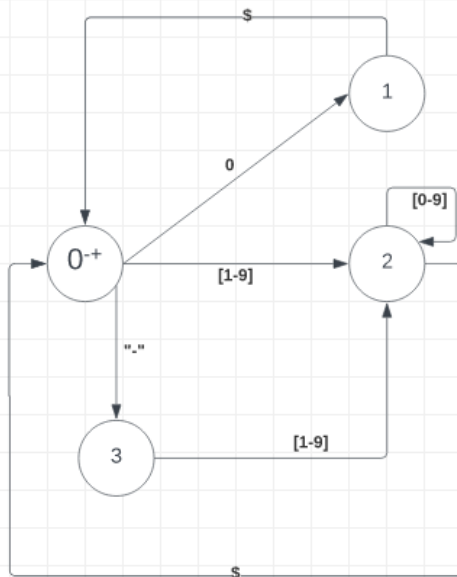
```
--- Palabra regular: 0$-78$88$075$0xAFF1$ Estado: true---  
--- Palabra regular: 0$7-8$88$075$0xAFF1$ Estado: false---  
--- Palabra regular: 0$7-8$88$075$AFF1$ Estado: false---  
--- Palabra regular: 0750xAFF1$ Estado: false---  
--- Palabra regular: 0x$88$099 Estado: false---  
--- Palabra regular: 0x$88$077 Estado: false---  
--- Palabra regular: 0x$88$077$ Estado: false---  
--- Palabra regular: 0x44A$88$077$ Estado: true---
```

ERX:

`/^((([Xx][1-9A-Fa-f][0-9A-Fa-f]*[01-7][0-7]*|-{0,1}[1-9][0-9]*|0x0|0c0|0)\$))*$/`

En el punto 1 decidimos hacer los tres autómatas por separado dado que consideramos que sería una forma más fácil de llevar a código, además de una forma más prolija de diagramar cada uno de los autómatas.

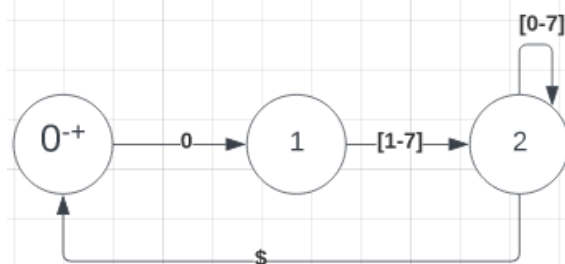
A continuación, los tres autómatas con sus respectivas TT:

Decimal:

TT	-		0	[1-9]	\$
0+		3	1	2	4
	1	4	4	4	0
	2	4	2	2	0
	3	4	4	2	4
	4	4	4	4	4

Hexadecimal:

TT		0	[Xx]	[A-Fa-f1-9]	\$
0-+		1	4	4	4
	1	4	2	4	4
	2	4	4	3	4
	3	3	4	3	0
	4	4	4	4	4

Octal:

TT	0	[1-7]	\$
0-+	1	4	4
1	4	2	4
2	2	2	0
4	4	4	4

Una vez hecho esto, pasamos a crearlos en C

```
//Automata Octal
int automataOctal(char *cadena){

    //TT
    size_t estados[][3] = {
        // 0,    [1-7],    $
        {1,     4,     4}, //0
        {4,     2,     4}, //1
        {2,     2,     0}, //2
        {4,     4,     4}  //4
    };

    //Declaro estado final
    size_t estado_final=0;
    //Declaro estado inicial
    size_t estado_inicial = 0;
    //inicio analizando la cadena
    size_t estado = estado_inicial;
    int i;
    char c;
    int largo_cadena=strlen(cadena);
    for(i=0;i<largo_cadena;i++){
        //Caracter donde está parado
        c=cadena[i];
        //Hace la transición respecto a cuál caracter del alfabeto se está procesando
```

En estos se puede apreciar la tabla de transiciones hecha en una matriz llamada “estados”, los estados inicial y final (donde ambos son 0), y creamos una variable “estado” la cual marca la posición en donde estamos en la tabla. Por último obtenemos el largo de la cadena y pasamos a leerla caracter a caracter.

```
        if(c=='0') estado = estados[estado][0];
        else if(c >= '1' && c <= '7') estado = estados[estado][1];
        else if(c == '$') estado = estados[estado][2];
        //Si no pertenece al alfabeto va directo a estado de rechazo
        else estado = 4;

        //Si se rechaza, ya salimos del autómata y lo declaramos inválido
        if(estado == 4) return -1;
    }

    //Si no acabó en un estado final, el autómata se rechaza por estar incompleto
    if(estado == estado_final) return 0;
    else return -1;
}
```

Esta parte varía según el autómata (hexadecimal, decimal y octal), pero la lógica es la misma. Cuando los if validan con algún carácter, se actualiza la fila a la que nos movemos en la tabla “estados” con la variable “estado”, en caso de que no reconozca algún carácter, nos movemos a la fila 4(rechazo), y al final comprobamos que el estado que salió del ciclo for sea un estado final(en este caso el 0), si es cierto, devolvemos 0, sino un -1.

Luego juntaremos los tres autómatas para reconocer si en alguno de los tres hubo error léxico, en una función llamada “obtenerNumeros”

Primero declaramos una variable llamada “cadena” dándole un espacio específico, le pedimos al usuario que nos ingrese la cadena de números separada por \$, y se la pasamos a la función que tiene los autómatas

```
int main(void){
    char * cadena=malloc(sizeof(10000));
    printf("Ingrese una cadena de números separados por '$':\n ");
    scanf("%s",cadena);
    obtenerNumeros(cadena);
    return 0;
}
```

En esta función declaramos los contadores de cada autómata y copiamos la cadena que se recibe en una variable “copia” a la cual vamos dividiendo con “strtok” en partes que terminen con \$, pero como esta función “agarra” todo lo que esté HASTA cierto carácter, luego de copiarlo en “analizar” tenemos que volverlo a agregar con “strcat”.

Mientras vaya reconociendo tokens, los va a ir mostrando en pantalla.

```
void obtenerNumeros(char *cadena) {

    int cont_dec=0;
    int cont_hex=0;
    int cont_oc=0;
    int error=-1;
    char copia[strlen(cadena) + 1];
    strcpy(copia, cadena); // Copia la cadena original para no modificarla

    char *token = strtok(copia, "$"); // Divide la cadena en tokens separados por '$'
    char *analizar=malloc(sizeof(token));
    while (token != NULL) {
        strcpy(analizar,token);
        strcat(analizar,"$");
        printf("Número: %s\n", analizar);
    }
```

Dentro del while tenemos una cadena de if en donde la cadena obtenida se manda a cada autómatas, y si uno de estos la reconoce, aumenta su contador. Una vez que sale de esta cadena, avanzamos a la siguiente parte de la expresión que termine con \$.

```
if(automataOctal(analizar)==0) cont_oc++;
else if(automataDecimal(analizar)==0) cont_dec++;
else if(automataHexadecimal(analizar)==0) cont_hex++;
else {
    error=0; //hubo error
    break;
}
token = strtok(NULL, "$"); // Obtiene el siguiente token
}
if(error==0) printf("Error Léxico!\tNo se reconoció el número.\n Finalizando Programa...\n");
else printf("Cadenas Reconocidas:\n\n Decimales:%d\n Hexadecimales:%d\n Octales:%d\n", cont_dec, cont_hex, cont_oc)
}
```

A continuación vemos un ejemplo de su funcionalidad, donde se pudo reconocer 3 cadenas decimales(-78, 88 y 0), una octal(075), y una hexadecimal(0xAFF1).

```
Ingrese una cadena de números separados por '$':
0$-78$88$075$0xAFF1$
Número: 0$
Número: -78$
Número: 88$
Número: 075$
Número: 0xAFF1$
Cadenas Reconocidas:

Decimales:3
Hexadecimales:1
Octales:1
```

Abajo está el mismo ejemplo pero demostrando que admite tanto mayúsculas como minúsculas.

```
Ingrese una cadena de números separados por '$':  
0$-78$88$075$0Xaff1$  
Número: 0$  
Número: -78$  
Número: 88$  
Número: 075$  
Número: 0Xaff1$  
Cadenas Reconocidas:  
  
Decimales:3  
Hexadecimales:1  
Octales:1  
PS D:\Universidad\UTN\2_Nivel\ssl\ssl-tp-automatas-g25>
```

Un ejemplo de cómo rechaza cadenas incorrectas es el siguiente, ya que el autómatas octal no admite 8 ni 9. El \$\$ es ignorado como se ve en el segundo.

```
Ingrese una cadena de números separados por '$':  
08976$$2375$0x4  
Número: 08976$  
Error Léxico! No se reconoció el número.  
Finalizando Programa...
```

Tampoco admite 0's a la izquierda sea para cualquier sistema:

```
Ingrese una cadena de números separados por '$':  
0045$-78$  
Número: 0045$  
Error Léxico! No se reconoció el número.  
Finalizando Programa...  
PS D:\Universidad\UTN\2_Nivel\ssl\ssl-tp-automatas-g25>
```

Otro caso de error, por ejemplo se ingresa una cadena que no es reconocida por ningún autómatas porque hay un carácter 'p' colado en la cadena.

```
Ingrese una cadena de números separados por '$':  
23p$0xADD$  
Número: 23p$  
Error Léxico! No se reconoció el número.  
Finalizando Programa...  
PS D:\Universidad\UTN\2_Nivel\ssl\ssl-tp-automatas-g25>
```

PUNTO 2

En el punto 2 (caracterNumericoAEntero) realizamos una función que dado un carácter numérico (del '0' al '9'), traduzca mediante un cálculo a un entero operable teniendo en cuenta los valores de la tabla ASCII.

```
/*Punto 2 - Pasar UN caracter numérico a un entero*/  
int caracterNumericoAEntero (const char caracter)  
{  
    if (caracter >= '0' && caracter <='9')  
        return caracter - '0'; //Recalibra el desplazamiento de la tabla ascii  
    else return -1; //Al solo tomar caracteres numéricos y no el signo, el número negativo nos informaría de un error.  
};  
You, 4 days ago • Los 3 puntos hechos :) ...
```

Tests:

- 1) Testeo número normal:

```
Ingrese un caracter numérico: 9  
9  
PS D:\Universidad\UTN\2_Nivel\ssl\ssl-tp-automatas-g25> █
```

- 2) Testeo numero mal ingresado. Debe dar un error

```
Ingrese un caracter numérico: $  
  
Error  
PS D:\Universidad\UTN\2_Nivel\ssl\ssl-tp-automatas-g25> █
```

PUNTO 3

En el punto 3 decidimos optar por usar el autómata decimal, modificándolo brevemente en la función `expresionValida` (hacemos una copia de la expresión, en esa copia reemplazamos los operadores por "\$", y lo validamos con el autómata decimal), con la cual podemos verificar que la composición de la expresión matemática dada sea correcta.

En la función principal pedimos que se ingrese una cadena que contenga una expresión matemática, y esta es validada por la función "`expresionValida`". En caso de pasar este control, se ejecutará "`evaluarExpresión`".

```
int main() {  
    char expresion[100]; //Se pueden realizar operaciones de hasta 100 caracteres  
  
    /*El programa se mantendrá activo hasta ingresar una expresión errónea*/  
    while(1){  
        printf("Ingrese una expresión matemática sin paréntesis: ");  
        scanf("%s", expresion);  
        if (expresionValida(expresion) == -1)  
        {  
            printf("Cadena invalida para este formato. ");  
            return 0;  
        }  
        else {  
            int resultado = evaluarExpresion(expresion);  
            printf("El resultado es: %d\n", resultado);  
        }  
    }  
    return 0;  
}
```

`ExpresionValida` lee la cadena y reemplaza los símbolos de suma, resta, mul y div, por \$, para luego pasarle esta cadena al automata decimal que hicimos en el punto 1

```
int expresionValida (const char *expresion) {  
    char *cadena=malloc(sizeof(cadena));  
    strcpy(cadena, expresion);  
    strcat(cadena,"$"); //Ponemos '$' al final de la cadena para que la valide el autómata  
    int longitud = strlen(cadena);  
    /*Reemplazamos todos los operadores por '$'*/  
    for (int i = 0; i < longitud; i++) {  
        if (cadena[i] == '+' || cadena[i] == '-' || cadena[i] == '*' || cadena[i] == '/') {  
            cadena[i] = '$';  
        }  
    }  
    /*Retornamos la cadena formateada procesada por el autómata*/  
    return automataDecimal(cadena);  
};
```

Luego de que la cadena está validada, se utiliza el algoritmo (evaluarExpresion) para resolver una cadena de chars, dividiéndolos entre dos arrays, una de “numeros” (utilizando un algoritmo similar al del punto 2) y otra de “operadores”.

```
int evaluarExpresion(const char *expresion) {
    int longitud = strlen(expresion);
    int numeros[100]; // Suponemos que la expresión no tiene más de 100 números
    char operadores[100]; // Suponemos que la expresión no tiene más de 100 operadores
    int top_num = -1; // Inicializamos el índice del arreglo de números
    int top_op = -1; // Inicializamos el índice del arreglo de operadores

    for (int i = 0; i < longitud; i++) {
        if (isdigit(expresion[i])) {
            // Si el carácter es un dígito, se convierte en un número y se guarda en la pila
            int numero = 0;
            while (i < longitud && isdigit(expresion[i])) {
                numero = numero * 10 + caracterNumericoAEntero(expresion[i]);
                i++;
            }
            numeros[++top_num] = numero;
            i--; // Retroceder el índice para evitar perder el siguiente carácter
        } else if (esOperador(expresion[i])) {
            // Si el carácter es un operador, maneja la precedencia
            while (top_op >= 0 && precedencia(operadores[top_op]) >= precedencia(expresion[i])) {
                char operador = operadores[top_op--];
                int segundoNumero = numeros[top_num--];
                int primerNumero = numeros[top_num--];

                // Realiza la operación y guarda el resultado en el arreglo de números
                switch (operador) {
```

para luego establecer la precedencia con un switch, y hacemos lo mismo para realizar las operaciones

```
                switch (operador) {
                    case '+':
                        numeros[++top_num] = primerNumero + segundoNumero;
                        break;
                    case '-':
                        numeros[++top_num] = primerNumero - segundoNumero;
                        break;
                    case '*':
                        numeros[++top_num] = primerNumero * segundoNumero;
                        break;
                    case '/':
                        if (segundoNumero != 0) {
                            numeros[++top_num] = primerNumero / segundoNumero;
                        } else {
                            printf("Error: División por cero\n");
                            exit(1);
                        }
                        break;
                }
            }
            operadores[++top_op] = expresion[i];
        } else return errno;
    }
}
```

En el caso de que se ingrese una cadena inválida, se mostrará un mensaje de error.

Decidimos tomar como límite una operación matemática de hasta 100 caracteres para simplificar el código, ya que consideramos innecesaria y fútil la

evaluación de una operación matemática simple más larga.

Tests:

- 1) Testeo operación descrita en el TP ($3+4*8/2+3-5$). Debe dar 17.
- 2) Otro testeo precedencia (Operación $3+4*5$). Debe dar 23.
- 3) Testeo cadena errónea (doble operador. Ej: $3++5$). Debe dar error.

Debajo se muestran estos tres casos juntos:

```
Ingrese una expresión matemática sin paréntesis: 3+4*8/2+3-5
El resultado es: 17
Ingrese una expresión matemática sin paréntesis: 3+4*5
El resultado es: 23
Ingrese una expresión matemática sin paréntesis: 3++5
Cadena invalida para este formato.
PS D:\Universidad\UTN\2_Nivel\ssl\ssl-tp-automatas-g25> █
```

- 4) Testeo caracter invalido (Ej: $3+4\{5$). Debe dar error:

```
Ingrese una expresión matemática sin paréntesis: 3+4{5
Cadena invalida para este formato.
PS D:\Universidad\UTN\2_Nivel\ssl\ssl-tp-automatas-g25> █
```