

Universidad de San Carlos de Guatemala  
Facultad de Ingeniería  
Escuela de Ciencias y Sistemas  
Organización de Lenguajes y Compiladores 2  
Primer Semestre 2016



**Catedráticos:** Ing. Bayron López, Ing. Edgar Sabán

**Tutores académicos:** Julio Flores, Andrea Alvarez, Jordy González, Kevin Lajpop

# D-LEX

Primera práctica de laboratorio

## 1 OBJETIVOS:

---

### 1.1 OBJETIVO GENERAL

- Construir una solución de software que permita generar analizadores léxicos y visualizarlos interactivamente.

### 1.2 OBJETIVOS ESPECÍFICOS

- Aplicar el método de Thompson y cerradura para transformar expresiones regulares en Autómatas Finitos Deterministas (AFD).
- Generar reportes de errores léxicos
- Mostrar de forma gráfica los estados del AFD y el cambio de los mismos al reconocer un lexema.
- 

## 2 DESCRIPCIÓN GENERAL

---

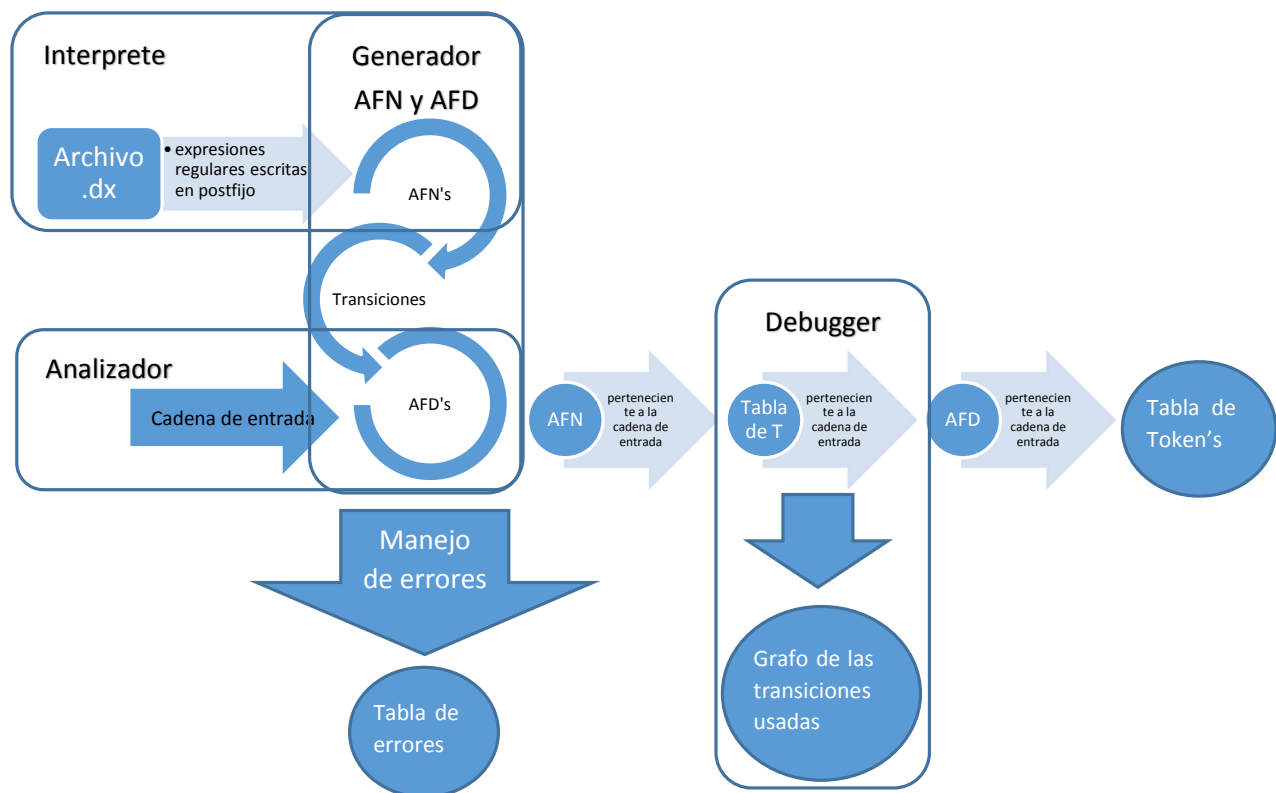
El estudio de los compiladores está dividido en tres cursos del área de Ciencias de la Computación: Lenguajes Formales y de Programación, Organización de Lenguajes y Compiladores 1 y Organización de Lenguajes y Compiladores 2. En cada uno de estos cursos se ha estudiado las diferentes etapas del compilador, uno de los objetivos del curso compiladores 2 es unificar todos estos conocimientos previos para comprender la segunda fase del compilador, la fase de síntesis.

La primera fase del proyecto consta de una práctica de laboratorio, en la cual se consolidarán los conceptos de análisis léxico. Para ello, se requiere la construcción de una aplicación que permita analizar y debuggear cadenas de entrada que serán aceptadas mediante expresiones regulares.

Dicha aplicación estará separada en cuatro partes:

- **Intérprete** módulo de la aplicación que analizará un archivo que contendrá expresiones regulares y sus reglas asociadas. El intérprete irá leyendo expresión regular por expresión regular, para aceptar un lexema dentro del lenguaje posteriormente definido.
- **Generador de AFN y AFD** módulo que realizará el análisis léxico, el cual recibirá la expresión regular que haya aceptado el lexema y construirá un grafo AFD asociado por medio del método de Thompson + cerradura.
- **Analizador** módulo que analizará una cadena grande de entrada y generara tabla de tokens o tabla de errores encontrados.
- **Debugger** módulo que permitirá analizar una cadena pequeña visualizando paso por paso el recorrido de la tabla de transiciones de forma gráfica.

## 2.1 GRÁFICA DEL FUNCIONAMIENTO DE LA APLICACIÓN:



### 3 CARACTERÍSTICAS DE LA APLICACIÓN

---

Se le atribuye desarrollar una aplicación en lenguaje C# que permita reconocer los archivos de entrada, utilizando la herramienta Irony, y sea capaz de realizar los análisis correspondientes para poder generar un debugger funcional.

Se requiere la construcción de editores de texto que sean capaces de proveer un entorno cómodo para la edición de expresiones regulares y cadenas de entrada. Además debe construirse un debugger que permita visualizar de manera gráfica la aceptación o denegación de cadenas, paso por paso o de manera automática.

La aplicación debe ser capaz de mostrar información de los errores encontrados para facilitar su solución.

La aplicación contará con 4 funcionalidades principales, las cuales están detalladas a continuación:

### 4 INTÉRPRETE

---

Utilizando la herramienta Irony se debe implementar un intérprete para la fase de análisis léxico y sintáctico de los archivos de entrada, que contendrán el conjunto de expresiones regulares.

El intérprete reconocerá sentencia por sentencia. En dichas sentencias, se encapsula el identificador del token, la expresión regular asociada y el símbolo que retornará. Cuando termine de realizar el método continuará con la siguiente expresión regular.

#### 4.1 DEFINICIÓN DE ENTRADA

La herramienta generadora de analizadores léxicos debe utilizar un lenguaje (DleX), dentro de éste se definen todas las representaciones léxicas que el analizador debe reconocer como propias del lenguaje fuente.

El archivo para definir las expresiones léxicas se compone de una única sección, en la cual se definirá el token que el intérprete deberá reconocer seguido de la definición de la expresión regular a las que corresponde a cada token y por último al símbolo que se retornará al reconocer un token.

## 4.2 DEFINICIÓN DE LENGUAJE DLEX

El archivo fuente del lenguaje que contiene las expresiones regulares tendrá la siguiente estructura:

```
%%  
  
sentencia1;  
  
sentencia2;  
  
sentencia3;  
  
sentencia4;  
  
%%
```

*Tabla 1: Estructura de archivo de expresiones regulares*

Los delimitadores que le indicarán al intérprete el inicio y final del archivo serán el doble porcentaje (%%), así mismo las sentencias estarán delimitadas al final por un punto y coma (;), la estructura de cada sentencia se define más adelante.

### 4.2.1 Sentencias

Cada sentencia deberá de contar con tres partes como se mencionó anteriormente, la estructura de una sentencia será la siguiente:

```
%%  
  
Id -> EXPRESION_REGULAR_EN_POSTFIJO -> RETORNO->PALABRAS_RESERVADAS;  
Num -> EXPRESION_REGULAR_EN_POSTFIJO->RETORNO->PALABRAS_RESERVADAS;  
If -> EXPRESION_REGULAR_EN_POSTFIJO -> RETORNO;  
While ->EXPRESION_REGULAR_EN_POSTFIJO -> RETORNO;  
  
%%
```

*Tabla 2: Estructura de sentencias*

Nota: como se visualiza en el ejemplo anterior, puede o no puede venir reglas asociadas para palabras reservadas.

#### 4.2.2 Expresiones regulares

Una expresión regular es la forma más óptima para establecer el patrón que representa al token. Las expresiones regulares para utilizar en el generador y debugger de analizadores léxicos se presentan en notación postfijo.

Las expresiones regulares tendrán la siguiente convención:

#### 4.2.3 Conjuntos

Símbolo	Definición
a b .	Concatenación.
a b	Disyunción
a~c	Conjunto, de 'a' o 'b' o 'c'
a~z	Conjunto de 'a' a la 'z', minúsculas.
A~Z	Conjunto de 'A' a la 'Z', mayúsculas.
0~7	Conjunto numérico, en este caso de '0' a '7'
0,2,4,6,8	Conjunto de dígitos pares.
a,b,c	Conjunto de letras a, b o c.
!~&	Conjunto de caracteres que están entre ! (33 en código ascii) y & (38 en código ascii).

*Tabla 3: Conjuntos*

Cuando se desee definir un conjunto, no es necesario poner el retorno ni una expresión regular asociada, simplemente se define el conjunto con la palabra reservada **"CONJ"**, tal como se visualiza más adelante en el ejemplo.

#### 4.2.4 Cuantificadores

Símbolo	Definición
*	indica 0 ó más
+	indica 1 ó más
?	indica 0 ó 1

*Tabla 4: Cuantificadores unarios*

#### 4.2.5 Caracteres especiales y macros propios del lenguaje

Símbolo	Definición
\n	Salto de línea.
\'	Comilla simple
\''	Comilla doble
\t	Tabulación
[:blanco:]	Representa, espacios en blanco y salto de línea.
[:todo:]	Todos los caracteres exceptuando el salto de línea (\n)

*Tabla 5: Caracteres propios*

#### 4.2.6 Ejemplos de expresiones regulares individuales

Postfijo	Infijo
letra -> a~z;	letra -> a~z;
digito -> 0~9;	digito -> 0~9;
numero -> 0~9+;	numero -> [0~9]+;
ID -> a~z a~z 0~9   ' _ ' *.	ID -> [a~z]([a~z]   [0~9]   ' _ ')*
DECIMAL->0~9+'.' 0~9+.	DECIMAL->{ [0~9]+}'.'{ [0~9]+}

*Tabla 6: Ejemplos de expresiones regulares individuales*

#### 4.2.7 Ejemplo de expresiones regulares en notación postfijo

```
%%  
CONJ: letra -> a~z;  
CONJ: digito -> 0~9;  
Id -> letra letra digito | " _ " * . -> RETORNO;  
CONJ: abc -> a,b,c;  
Num -> digito + -> RETORNO;  
Dec -> digito + "." . digito + . -> RETORNO;  
%%
```

*Tabla 7: Estructura de expresiones regulares*

#### 4.2.8 Reglas y retorno

Dentro de esta parte se establecerán las acciones que realizará el intérprete al reconocer un token determinado, en esta parte se utilizarán dos métodos retorno y un método error, también se podrá acceder a tres atributos del analizador:

- yytext de tipo string, que contendrá el texto del token que se ha reconocido.
- yyline de tipo int, que contendrá la línea del token que se ha reconocido.
- yyrow de tipo int, que contendrá la columna del token que se ha reconocido.

##### 4.2.8.1 Descripción de los métodos:

- Método retorno(token, línea, columna)
  - Este método recibe como parámetro un token, el cual agregara a la lista de tokens reconocidos, este método se utiliza para registrar las **palabras reservadas** de un lenguaje dado que carecen de valor o un tipo.
- Método retorno(token, valor, tipo, línea, columna)
  - Este método recibe como parámetro un token y su respectivo valor los cuales agregara a la lista de tokens reconocidos, en este método se pueden definir **identificadores, valores** y todo lo que tenga o pueda tener un tipo y/o un valor asociado.
- Método error(línea, columna, valor)
  - Este método se utilizará para manejar errores dentro del analizador léxico, recibe como parámetros columna, fila y valor. El método de error solamente debe de venir **una vez** dentro del archivo que contiene expresiones regulares.

#### 4.2.9 Tipos

Se deben manejar distintos tipos de datos en el analizador léxico, los tipos de datos que se manejan serán los siguientes:

- string
- char
- int
- float
- bool

##### 4.2.10 Ejemplo completo de un archivo de entrada de expresiones regulares

```
%%  
CONJ: letra-> a~z;
```



```

CONJ: digito->0~9;

Id -> letra letra digito | “_” | * . -> retorno(id, yytext, string, yyline, yyrow) ->
RESERV->[“If” -> retorno(if, yyline, yyrow)];

Num -> digito + -> retorno(Num, yytext, int, yyline, yyrow);

Dec -> digito + “.” . digito + . -> retorno(Dec, yytext, float, yyline, yyrow);

Mas -> “+” -> retorno(Mas, yyline, yyrow);

%%

```

*Tabla 8: Estructura final de expresiones regulares*

#### 4.2.11 Ejemplo 2:

```

%%

CONJ: letra-> a~z;

Id -> letra letra digito | “_” | * . -> retorno(id, yytext, string, yyline, yyrow)-
>RESERV[“If” -> retorno(if, yyline, yyrow); “while”->retorno(while, yyline, yyrow)];

CONJ: digito->0~9;

CONJ: carac->!~&;

Num -> digito + ->retorno(Num, yytext, int, yyline, yyrow);->RESERV[“6”->retorno(seis,
yyline, yyrow)];

Dec -> digito + “.” . digito + . -> retorno(Dec, yytext, float, yyline, yyrow);

Str -> letra + “\” .\” . ->retorno(Str, yytext, string, yyline, yyrow);

Chr -> letra “\” .\’ . ->(Chr, yytext, string, yyline, yyrow);

Mas -> “+” -> retorno(Mas, yyline, yyrow);

Tab-> \t ->retorno(tab, yyline, yyrow);

error-> [:todo:] ->error(yyline, yyrow, yytext);

%%

```

*Tabla 9: Ejemplo más completo*

Anotaciones:

- Se observa que la definición de **conjuntos** (CONJ) puede venir en cualquier parte del archivo.
- El método de **error** (error) solamente viene una vez en el archivo.
- Si en la entrada a analizar tenemos la palabra “If” responderá a la expresión regular de “Id” pero esta expresión regular tiene reglas de **palabras reservadas** asociadas (RESERV) por lo cual irá a buscar en estas reglas si cumple con alguna, en este caso la palabra “If” cumple con una regla de palabra reservada, por lo que ejecuta la acción de retornar únicamente el token, yyline y yyrow.
- La jerarquía en el reconocimiento de los tokens está dado por el orden en que son colocados.

### 4.3 SALIDA DE TOKENS

Al momento de que el debugger termine de realizar el análisis interactivo y de aceptar las cadenas en función de las expresiones regulares se generará una salida la cual será utilizada por el analizador de sintaxis en la siguiente fase, el cual será un archivo xml con la siguiente estructura:

```
<tokens>
  <token>
    <nombre>nombretoken1</nombre>
    <tipo>tipotoken1</tipo>
    <valor>valortoken1</valor>
    <yyline>valorlinea</yyline>
    <yyrow>valorcolumna</yyrow>
  </token>
  <token>
    <nombre>nombretoken2</nombre>
    <tipo>tipotoken2</tipo>
    <valor>valortoken2</valor>
    <yyline>valorlinea</yyline>
    <yyrow>valorcolumna</yyrow>
  </token>
  <token>
    <nombre>nombretoken3</nombre>
    <tipo>tipotoken3</tipo>
    <valor>valortoken3</valor>
    <yyline>valorlinea</yyline>
    <yyrow>valorcolumna</yyrow>
  </token>
</tokens>
```


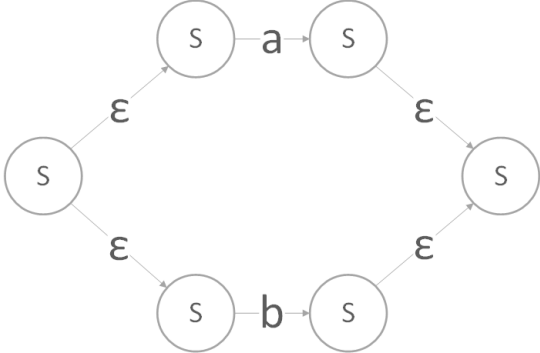
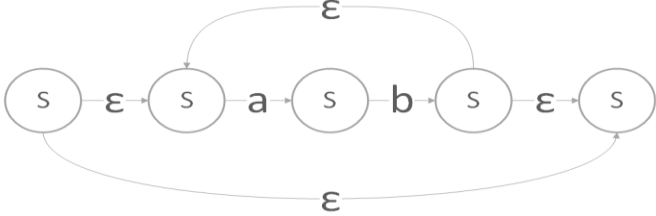
## 5 GENERADOR AFN Y AFD

Se contará con un generador de grafos que tiene como objetivo transformar expresiones regulares a autómatas finitos deterministas, y de esta manera validar un lexema dentro del lenguaje. Para ello se utilizará el método de Thompson y el cálculo de cerradura.

### 5.1 MÉTODO DE THOMPSON

El método de Thompson se basa en la creación de un Autómata Finito No-Determinista el cual se constituye de Estados y Transiciones. Dada una expresión debe transformarse de la siguiente manera:

Creación de Autómata Finito No-Determinista

Expresión	Autómata
<b>ab.</b>	
<b>ab </b>	
<b>a*</b>	
<b>a+</b>	<b>aa*.</b>
<b>a?</b>	<b>εa </b>

## 5.2 OPERACIÓN CERRADURA

Debido a que es necesario convertir el Autómata Finito No-Determinista en uno Determinista se aplican las siguientes operaciones:

### Cerradura(T):

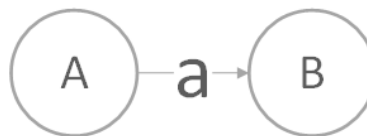
Conjunto de estados alcanzables desde el conjunto de estados "T" utilizando solamente transiciones  $\epsilon$ .

### Mover(T,a):

Conjunto de estados alcanzables desde el conjunto de estados "T" utilizando transiciones "a".

## 5.3 PROCEDIMIENTO:

1. Se aplica la operación sobre el estado inicial del AFN.  
 $Cerradura(0)=A$   
El conjunto de estados resultantes debe nombrarse, en este ejemplo lo llamaremos el conjunto "A".
2. Debe crearse un nuevo conjunto de estados para cada terminal "a" de la forma:  
 $Cerradura(Mover(A,a))=B$   
Este nuevo conjunto deberá de nombrarse por ejemplo "B". Si ya existe un conjunto de estados idéntico al resultado de la operación se le coloca el mismo nombre.
3. Se repite el paso 2 para cada conjunto de estados que se genere como resultado del mismo. Este paso acaba cuando ya no hay más conjuntos nuevos a los que aplicar el paso 2.
4. Se toman los conjuntos de estados como los nuevos estados del AFD y para cada operación Mover aplicada se utiliza como una transacción en la Tabla de Transiciones. Cada conjunto de estados que posea al último estado del método de Thompson se convierte en un estado de aceptación.

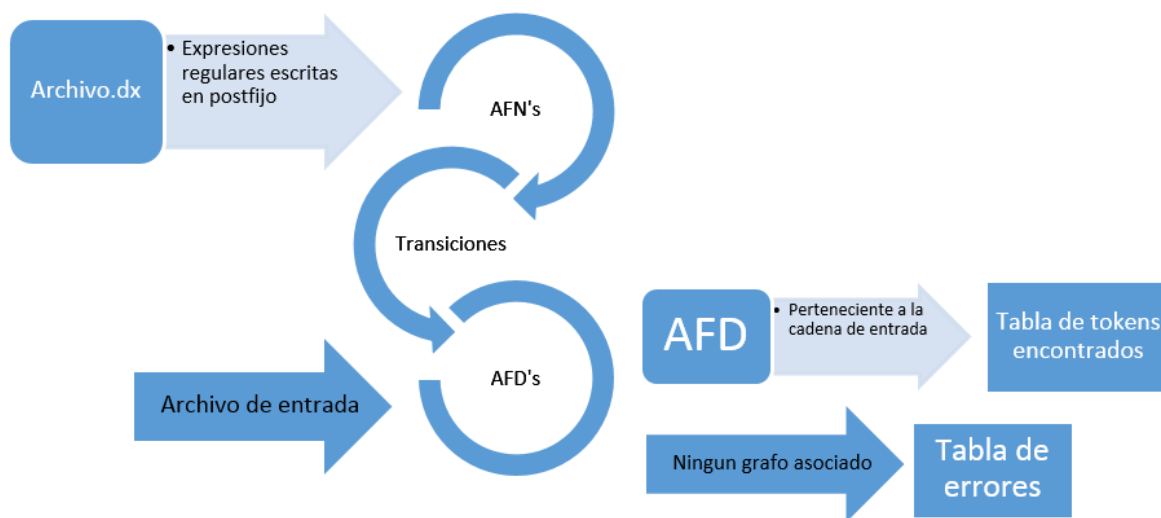


El módulo deberá generar tanto el AFN como el AFD utilizando la herramienta Graphviz para poder mostrarlos en un área junto con la tabla con las operaciones de cerradura y la tabla de transiciones de cada lexema analizado. En las gráficas del AFN y AFD se debe mostrar los estados del autómata, el estado inicial, los estados de aceptación y las transiciones del mismo.

## 6 ANALIZADOR

---

Esta funcionalidad tiene como objetivo analizar cadenas de entrada grandes. Para analizar una cadena es necesario cargar un archivo .dx con las expresiones regulares y realizar todo el procedimiento de generación del AFD por medio del método de Thompson + cerradura, para la aceptación de lexemas dentro del código fuente definido por las expresiones. Al terminar el análisis se debe de generar la lista de tokens encontrados o la lista de errores con su respectiva descripción.

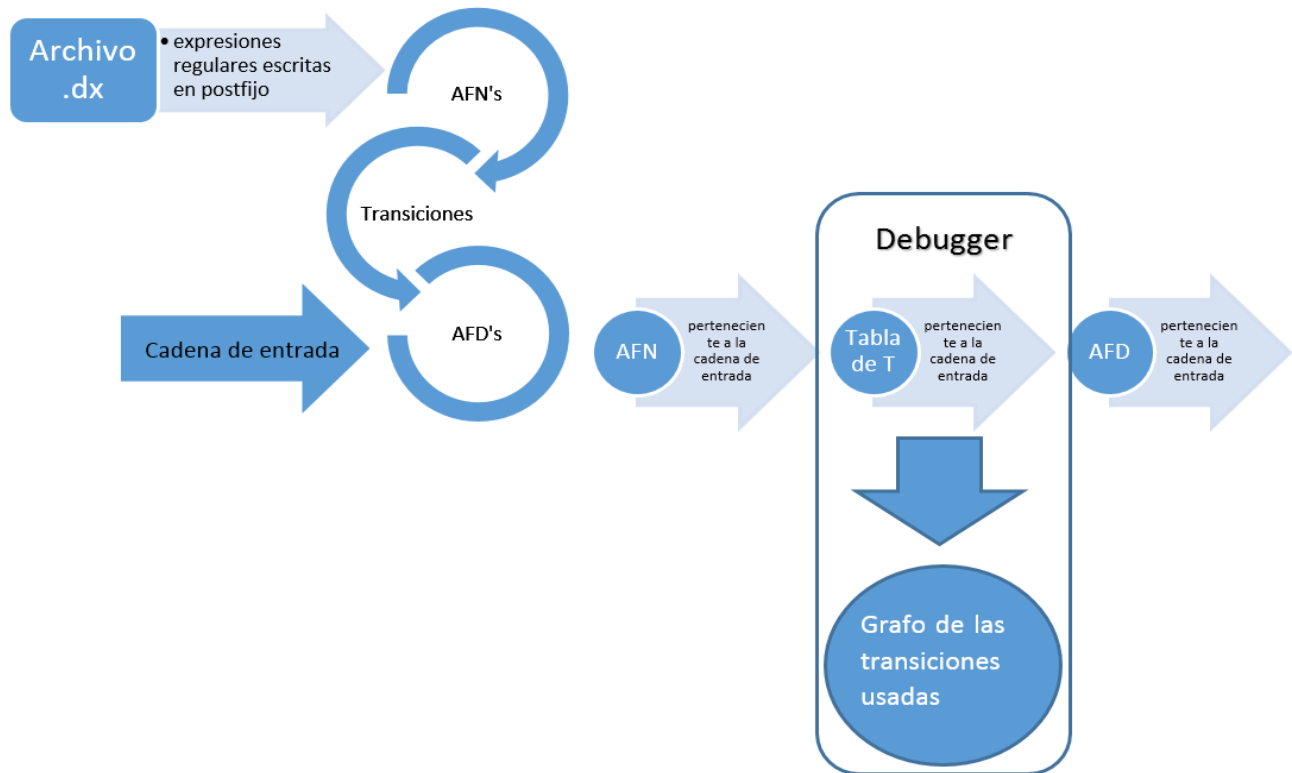


## 7 DEBUGGER

---

Esta funcionalidad tiene como objetivo visualizar el recorrido de la tabla de transiciones de forma gráfica. Para ello, se debe haber cargado previamente un archivo .dx que contiene las expresiones regulares en postfijo y de esta manera poder construir el autómata finito no determinista (AFN) por medio del método de Thompson, crear tabla de transiciones con el método de la cerradura y generar el autómata finito determinista (AFD) para la expresión regular en general. Una vez cargadas las estructuras se ingresa la cadena de entrada y se podrán visualizar las transiciones (una por una) utilizadas para aceptar la cadena de entrada, en dado caso la cadena de entrada sea errónea se visualizará en que transición hubo error (esto es un reporte de error gráfico).

### Proceso interno del debugger



#### 7.1 MODALIDADES DEL DEBUGGER:

Se tiene dos modalidades para el debugger, la primera será debugger paso a paso donde el programa mostrará el autómata usado para que la cadena sea aceptada, transición por transición, es decir cada vez que se presione en el botón next, este mostrará una nueva transición o paso hasta que la cadena sea aceptada o rechazada.

La segunda modalidad será debugger automático, donde se cambia el parámetro de tiempo por medio de un slider y se ejecutará por medio de un botón run, y se formará el autómata transición por transición de manera automática, refrescándose la imagen de manera automática con respecto al parámetro de tiempo ingresado, deteniéndose hasta que la cadena sea aceptada o rechazada.

## 7.2 FUNCIONAMIENTO:

Después de ingresar el archivo .dx (dependiendo de cuántas expresiones regulares se ingresen el analizador creará varios AFN, tablas de transiciones y AFD que pertenecen a cada una de las expresiones regulares ingresadas en dicho archivo) y luego de que estén cargados en memoria todos los autómatas y tablas, debe ingresarse una cadena de entrada (.txt) para comenzar el proceso de debugger y el programa tendrá que escoger un AFD con el cual la cadena será aceptada, el programa tendrá que mostrar el AFN y la tabla de transiciones a la cual pertenece dicho AFD. Por medio de los controles explicados anteriormente se recorrerá la tabla de transiciones creando un nuevo autómata (en la sección del flujo AFD de la interfaz de ejemplo) mostrando todas las transiciones utilizadas (paso por paso o de manera automática) hasta que la cadena sea aceptada.

# 8 DISEÑO DE LA INTERFAZ

---

La aplicación debe contar con una interfaz amigable e intuitiva para que el usuario final interaccione con la aplicación. Esta deberá contener todas las características para poder brindar comodidad al usuario. Asimismo contará con editores de texto en los cuales se podrán añadir los archivos y/o realizar la escritura de expresiones regulares o cadenas de entrada, sobre los que se realizarán los análisis necesarios para obtener la salida de tokens.

La descripción gráfica mostrada a continuación es con fines ilustrativos, y queda a consideración del estudiante su implementación.

## 8.1 BARRA DE MENÚ

Deberá contener como mínimo los siguientes menús:

### 8.1.1 Archivo

- **Nuevo:** Despliega un nuevo documento en blanco para editar.
- **Abrir:** Permite abrir un archivo .dx y .txt
- **Guardar:** Permite guardar el programa editado actualmente con extensión .dx y .txt
- **Buscar:** Permitirá realizar búsquedas exactas dentro del área de texto.
- **Salir:** Con esta opción se cerrará la aplicación.

### 8.1.2 Analizador

- **Cargar:** Mostrará un área de texto para cargar las expresiones *regulares*.
- **Analizar:** Mostrará un área de texto para ingresar la cadena de *entrada que se desea analizar*.
- **Debugger:** Mostrará un área para debuggear una cadena de entrada pequeña.

### 8.1.3 Reportes

- **Tabla de tokens:** Mostrará todos los tokens reconocidos en la entrada.
- **Errores:** Mostrará una tabla de errores léxicos encontrados en la entrada.

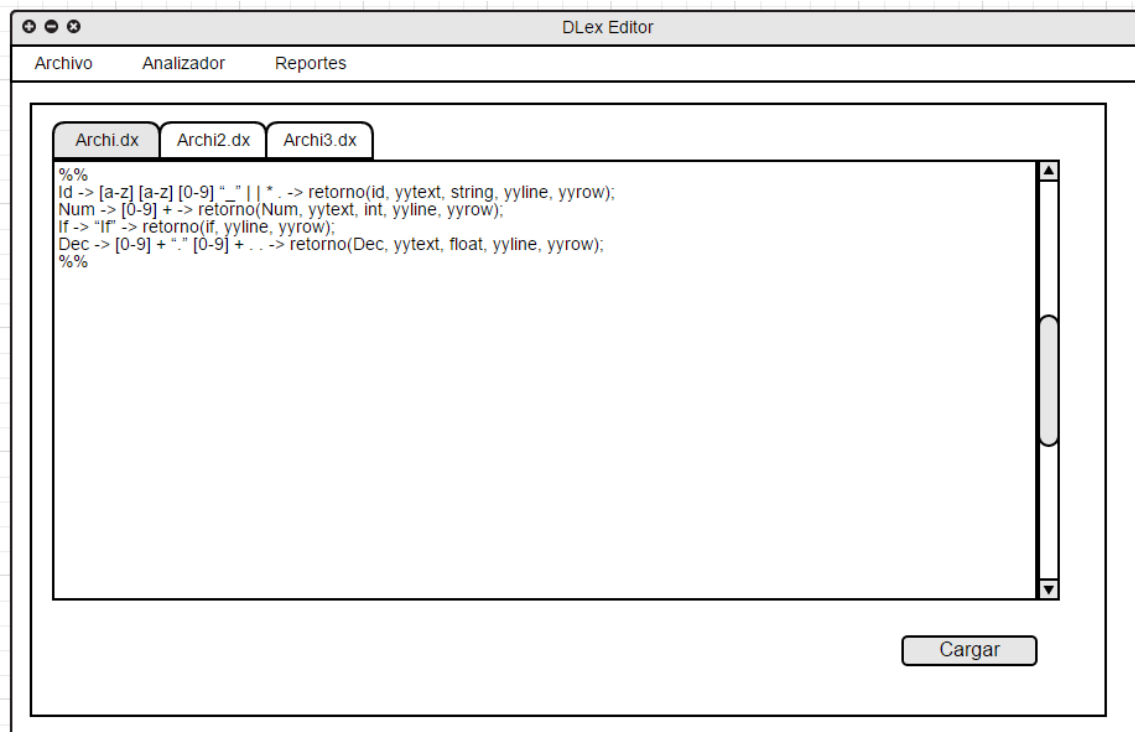
*Nota: Para que la aplicación sea más eficiente se necesita que el editor de texto pueda tener abierto más de un archivo a la vez.*

## 8.2 EDITORES DE TEXTO

La interfaz gráfica contará con 2 editores de texto, uno para poder abrir y/o editar archivos .dx que contendrá las expresiones regulares, y otro para abrir y/o editar archivos .txt que contendrá la cadena de texto que se quiere analizar. Al abrir archivos se deben filtrar según las extensiones de cada archivo.

**Analizador -> Cargar:** deberá abrir un área de texto donde se puedan editar solo archivos con extensión .dx

### 8.2.1 Ejemplo interfaz para cargar expresiones regulares al analizador.

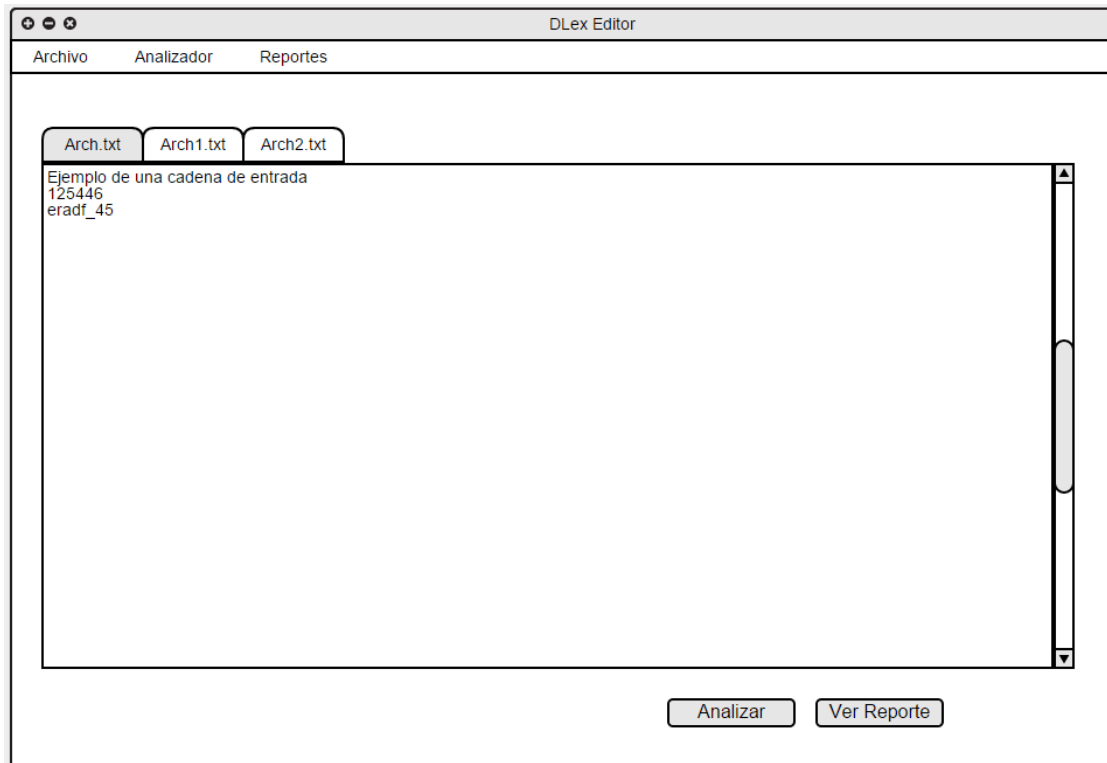


**Cargar:** cargará a memoria todas las expresiones regulares que están en el cuadro de texto actual.

**Analizador -> Analizar:** deberá abrir un área de texto donde se puedan editar solo archivos con extensión .txt



### 8.2.2 Ejemplo interfaz para analizar una cadena de entrada.

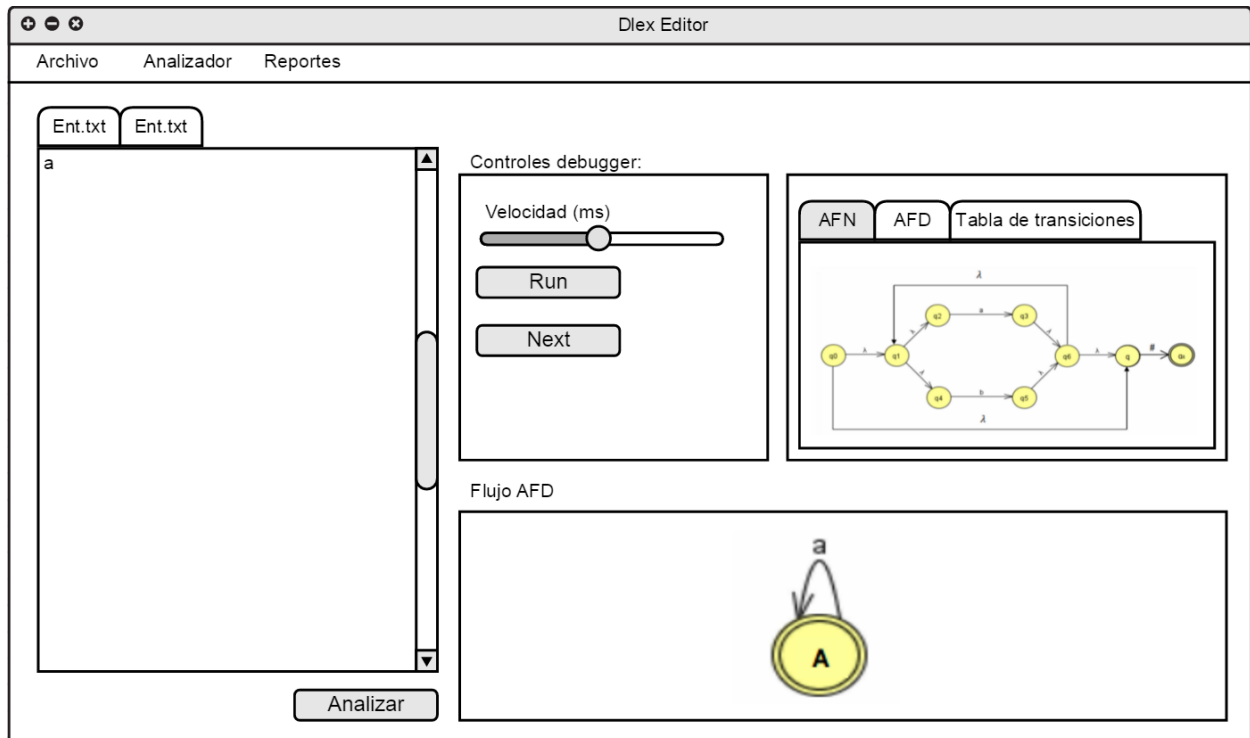


**Analizar:** Analizará el área de texto que estará.

**Ver Reporte:** Redireccionará al área de reportes de errores o tokens según sea el caso ocurrido al analizar la entrada.

**Analizador -> Debugger:** deberá mostrar un área de texto para poder analizar cadenas pequeñas, ver gráficas de AFN y AFD, y modificación de rapidez del debugger.

### 8.2.3 Ejemplo interfaz para la funcionalidad debugger:



## 8.3 TABLA DE TOKENS

Mostrará la tabla de tokens que fue creada por medio del análisis a los archivos de entrada. Debe estar en formato XML dentro de la aplicación. Debe poseer al menos nombre, tipo, valor, columna, fila.

## 8.4 REPORTE DE ERRORES

Mostrará los errores que fueron encontrados durante el análisis. Debe estar en formato XML dentro de la aplicación, mostrando el símbolo o lexema que provocó el error, fila, columna y una descripción.

Símbolo	Columna	Fila	Descripción

## 9 RESTRICCIONES

---

- La aplicación deberá ser desarrollada sobre el Microsoft .Net Framework, específicamente utilizando el lenguaje C#.
- Para graficar el AFD se utilizará GraphViz
- Para el análisis léxico y sintáctico de los archivos .dx se utilizará Irony
- La práctica es individual.
- Copias parciales o totales tendrán nota de 0 puntos y serán reportados a la Escuela de Ciencias y Sistemas.
- Se deberán utilizar los lenguajes y herramientas indicadas, en caso contrario no se tendrá derecho de calificación.
- No está permitido la modificación del código fuente.
- Para tener derecho a calificación deberán implementarse las siguientes funcionalidades:
  - Generar lista de tokens
  - Graficar AFD

## 10 ENTREGABLES

---

- Aplicación Funcional
  - Código fuente
  - Gramática escrita en Irony
  - Manual Técnico
  - Manual de Usuario
- 
- **Fecha de entrega:** Lunes 20 de febrero