Trabajo Práctico 1
Implementación de AFDs (v1.1)

Teoría de Lenguajes
1er cuatrimestre 2015

Fecha de entrega: 29 de abril

1. Objetivos

Este trabajo tiene como objetivos principales:

- Familiarizarse con la implementación de los algoritmos vistos en clase para el manejo de expresiones regulares y autómatas finitos (e.g., minimización e intersección).
- Ver las restricciones y dificultades de la implementación de cada algoritmo.
- Aplicar autómatas y expresiones regulares para resolver un problema simple.

2. Introducción

Se deberá diseñar e implementar un programa que permita la construcción y ejecución de Autómatas Finitos Determinísticos (AFDs). Este programa deberá correr como un comando por consola[1], cumpliendo al menos los ítems definidos en la siguiente sección.
En caso de que ocurra algún error el programa deberá retornar distinto de cero, y escribir el mensaje correspondiente en *standard error* con el detalle del mismo (e.g., 'El archivo de autómata no sigue el formato esperado').

3. Cosas a implementar

En líneas generales, se partirá de un archivo de texto con una expresión regular 'parseada', a partir de la cual habrá que generar un autómata finito determinístico, el cual deberá grabarse en otro archivo de texto siguiendo un formato determinado.

En todos los casos, los formatos para el contenido de los archivos de los autómatas y expresiones regulares pueden encontrarse en la Sección 4.

a) Dada un archivo que contiene una expresión regular parseada,

- generar un autómata finito correspondiente
- si no es determinístico, determinizarlo
- minimizar el autómata obtenido
- guardarlo en el archivo indicado.

```
./AFD -leng <archivo_regex> -aut <archivo_autómata>
```

[1] Deberá poder ejecutarse en las computadoras de los labos del DC.

b) Dado un autómata finito determinístico,

- decidir si una cadena pertenece a su lenguaje
- escribir `TRUE` o `FALSE`, según corresponda, por *standard output*.

```
./AFD -aut <archivo_autómata> <cadena>
```

c) Dado un autómata finito determinístico, generar el grafo correspondiente en lenguaje DOT[2] y grabarlo en el archivo indicado.

```
./AFD -aut <archivo_autómata> -dot <archivo_dot>
```

Si se quiere visualizar el archivo DOT, se puede generar un archivo de imagen con el comando `dot -Tps archivo.dot -o salida.ps` (o `-Tpng`, para archivos png).

También puede verse interactivamente con el comando `xdot archivo.dot`.

d) Dados dos autómatas determinísticos, obtener el autómata mínimo resultante para la **intersección** de sus lenguajes, y grabarlo en el archivo indicado:

```
./AFD -intersec -aut1 <archivo_autómata> -aut2 <archivo_autómata> -aut <archivo_autómata>
```

e) Dado un autómata finito determinístico, obtener el autómata mínimo resultante para el **complemento** de su lenguaje, y grabarlo en el archivo definido:

```
./AFD -complemento -aut1 <archivo_autómata> -aut <archivo_autómata>
```

f) Dados dos autómatas determinísticos, decidir si son **equivalentes**, escribiendo `TRUE` o `FALSE`, según corresponda, por *standard output*:

```
./AFD -equival -aut1 <archivo_autómata> -aut2 <archivo_autómata>
```

Consejo: aprovechar las funciones generadas para los ítems anteriores.

4. Formatos

Se incluyen los siguientes formatos:

- `<cadena>` : string a verificar su pertenencia al lenguaje (ver Sección 4.1).
- `<archivo_regex>` : archivo que contiene una expresión regular parseada (ver Sección 4.2).
- `<archivo_autómata>` : archivo que representa un autómata, que contiene los mismos elementos del formalismo visto en clase (ver Sección 4.3).

4.1. Cadenas a utilizar

Podría utilizarse los siguientes tipos de cadenas en la definición de las expresiones regulares y cadenas de prueba:

- El alfabeto a utilizar será un subconjunto de los carácteres alfanuméricos (mayúsculas y minúsculas, **sin** vocales con tilde ni eñes) y signos de puntación y otros (`[,:;.¿?!¡()"'\&-]`), espacios y tabs (`[ \t]`).
- El lenguaje a reconocer siempre será regular (i.e., no es necesario verificarlo en el programa).
- El formato de los archivos de texto es ASCII-compatible y el fin de línea es sólo con `'\n'`.

[2] The DOT Language: http://www.graphviz.org/doc/info/lang.html

4.2. Archivo de expresión regular

Las expresiones regulares quedarán representadas por la siguiente estructura de árbol dentro del archivo de las expresiones regulares (`<archivo_regex>`).

```
<regex>       :- símbolo_alfabeto | <operación>
<operación>   :- <operador>(\n\t<regex>)+
<operador>    :- {CONCAT}cant_operandos|{OR}cant_operandos|{STAR}|{PLUS}|{OPT}
```

Donde cada uno de los operadores representan los siguientes operadores de expresiones regulares:

- `{CONCAT}`: concatenación de dos o más expresiones
- `{STAR}`: clausura de Kleene (operador `*`)
- `{PLUS}` clausura positiva (operador `+`)
- `{OPT}`: opcionalidad (operador `?`, equivalente a (λ|EXP))
- `{OR}`: dos o más alternativas (operador `|`)

Ejemplo 1

Para la expresión regular `(a|b|c)*(de)+f`, el árbol correspondiente es el que se ve a continuación:

```
{CONCAT}3
	↦{STAR}
	↦	↦{OR}3
	↦	↦	↦a
	↦	↦	↦b
	↦	↦	↦c
	↦{PLUS}
	↦	↦{CONCAT}2
	↦	↦	↦d
	↦	↦	↦e
	↦f
```

Ejemplo 2

Para reconocer la expresión regular `(-ABC)?(0|1)+\t*`, de un número binario con un prefijo opcional y cero o más tabs al final, el árbol correspondiente es el que se ve a continuación:

```
{CONCAT}3
	↦{OPT}
	↦	↦{CONCAT}4
	↦	↦	↦-
	↦	↦	↦A
	↦	↦	↦B
	↦	↦	↦C
	↦{PLUS}
	↦	↦{OR}2
	↦	↦	↦0
	↦	↦	↦1
	↦{STAR}
	↦	↦\t
```

4.3. Archivo de representación del autómata

```
<nombre_estado>(\t <nombre_estado>)*                    % Estados del autómata (Q)
<simbolo_alfabeto>(\t <simbolo_alfabeto>)*              % Alfabeto del autómata (Sigma)
<nombre_estado>                                        % Estado inicial (q0)
(<nombre_estado>(\t <nombre_estado>)*)?                % Estados finales (F)
<nombre_estado>\t<simbolo_alfabeto>\t<nombre_estado>   % Definición de una transición
```

Ejemplo 1

En la Figura 1 se representa el autómata que reconoce el lenguaje denotado por la expresión regular `ac*(bf*)?`, como queda representado a continuación.

```
1  q0      →q1     →q2
2  a       →b      →c      →d      →e      →f
3  q0
4  q1      →q2
5  q0      →a      →q1
6  q1      →b      →q2
7  q1      →c      →q1
8  q2      →f      →q2
```

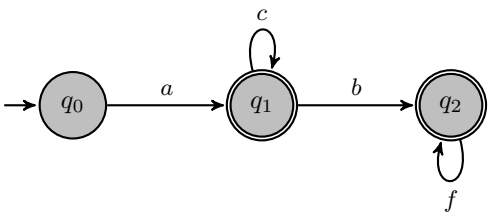

Figura 1: Ejemplo 1 de AFD

Ejemplo 2

Por otra parte, en la Figura 2 se representa el autómata que reconoce el lenguaje denotado por la expresión regular `(a*(ba*b)?(ccc)*`, como queda representado a continuación.

```
1  q0      →q1     →q2     →q3     →q4
2  a       →b      →c
3  q0
4  q0      →q4
5  q0      →a      →q0
6  q0      →b      →q1
7  q0      →c      →q2
8  q1      →a      →q1
9  q1      →b      →q0
10 q2      →c      →q3
11 q3      →c      →q4
12 q4      →c      →q2
```

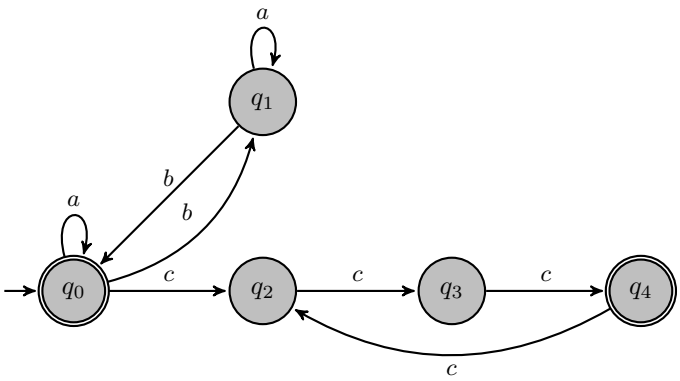

Figura 2: Ejemplo 2 de AFD

5. Caso de ejemplo

Dada la siguiente expresión regular parseada, correspondiente a `(a|b)(a|b)*`:

```
{CONCAT}2
        ↴{OR}2
        ↴       ↴a
        ↴       ↴b
        ↴{STAR}
        ↴       ↴{OR}2
        ↴       ↴       ↴a
        ↴       ↴       ↴b
```

se obtiene el siguiente autómata mínimo correspondiente:

```
q0      ↴q1
a       ↴b
q0
q1
q0      ↴a      ↴q1
q0      ↴b      ↴q1
q1      ↴a      ↴q1
q1      ↴b      ↴q1
```

Y una representación posible en DOT, en código y graficada, como se ve en la Figura 3.

```
strict digraph {
  rankdir=LR;
  node [shape = none, label = "", width = 0, height = 0]; qd;
  node [label="\N", width = 0.5, height = 0.5];
  node [shape = doublecircle]; q1;
  node [shape = circle];
  qd -> q0
  q0 -> q1 [label="a, b"]
  q1 -> q1 [label="a, b"]
}
```

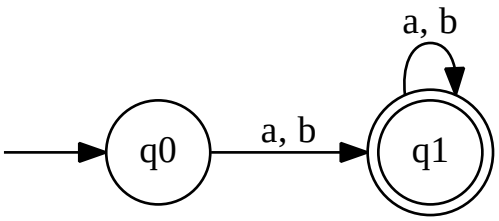

Figura 3: Diagrama resultante en DOT del caso de ejemplo

6. Detalles de la entrega

Pueden implementar su solución en Java, Python o C#. Si quieren usar otro lenguaje, primero consúltenlo con los docentes de TPs.
En cualquier caso, no pueden usarse bibliotecas/paquetes que sirvan para implementar autómatas, lenguajes o el procesamiento automático de expresiones regulares.

Se deberá enviar un e-mail conteniendo el código fuente y un informe breve (preferentemente, no más de 3 páginas + carátula) a tptleng@gmail.com.

La entrega debe cumplir con lo siguiente:

- El **asunto de mail** debe ser `[TL-TP1]` seguido por el nombre del grupo (e.g., "[TL-TP1] La banda de Kleene").
- El **código fuente** deberá estar documentado y entregarse en un archivo adjunto `.zip`, y **debe** incluir tests que permitan probar las funciones definidas, aparte de los brindados con el enunciado. Esto implica crear conjuntos de expresiones regulares, autómatas y listado de cadenas que debe y que no debe aceptar el autómata.
- El **informe** debe contener una carátula con el nombre del equipo e integrantes (LU, nombre y mail), y resumir las decisiones de diseño, referencia de algoritmos utilizados, problemas encontrados y algunas conclusiones sobre el trabajo realizado (e.g., restricciones de la implementación realizada).