Universidad de Granada

Ingeniería Informática

Practicas MC

Autor: José Antonio Ruiz Millán
Asignatura: Modelos de Computación
8 de enero de 2018

Índice

1. Practica 1: Optimizando Gramaticas

1.1. Enunciado

Determinar si la gramática $G = (\{S,A,B\}, \{a,b,c,d\}, P, S)$ donde P es el conjunto de reglas de produccion:

1. **S → AB**
2. **B → cB**
3. **A → Ab**
4. **B → d**
5. **A → a**

Genera un lenguaje de tipo 3.

1.2. Proceso

Lo primero que vamos a realizar será determinar el lenguaje **L** que genera la gramática anterior usando sus producciones.

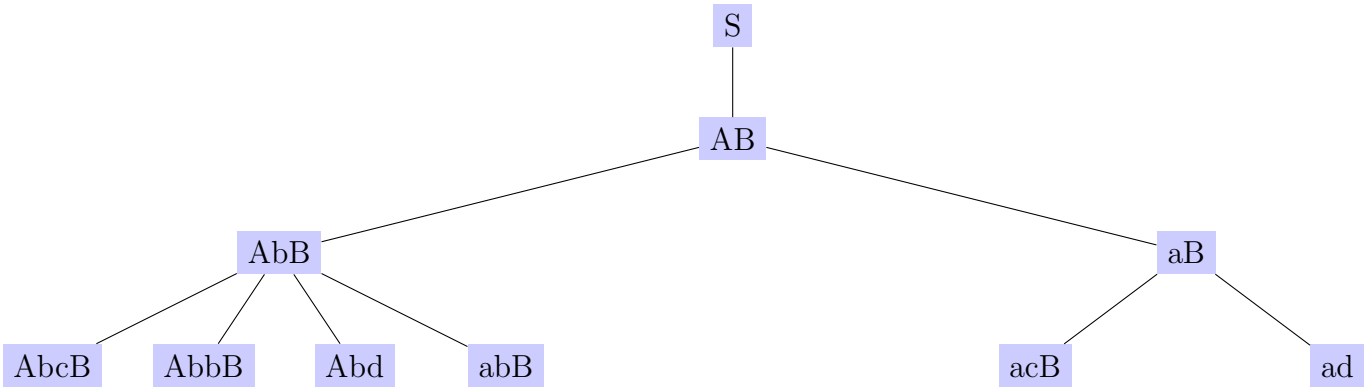
Empezaremos cambiando directamente las variables por los simbolos terminales, teniendo:

$\boldsymbol{S \rightarrow AB \rightarrow aB \rightarrow ad}$

Aplicando la produccion 1, 5 y 4 respectivamente.

Esta es la unica forma de eliminar las variables A y B, por lo que podemos ir viendo que la cadena que esta gramática va a producir será del estilo ***a ... d*** ya que las otras producciones nos mantienen la A y la B al principio y final respectivamente.

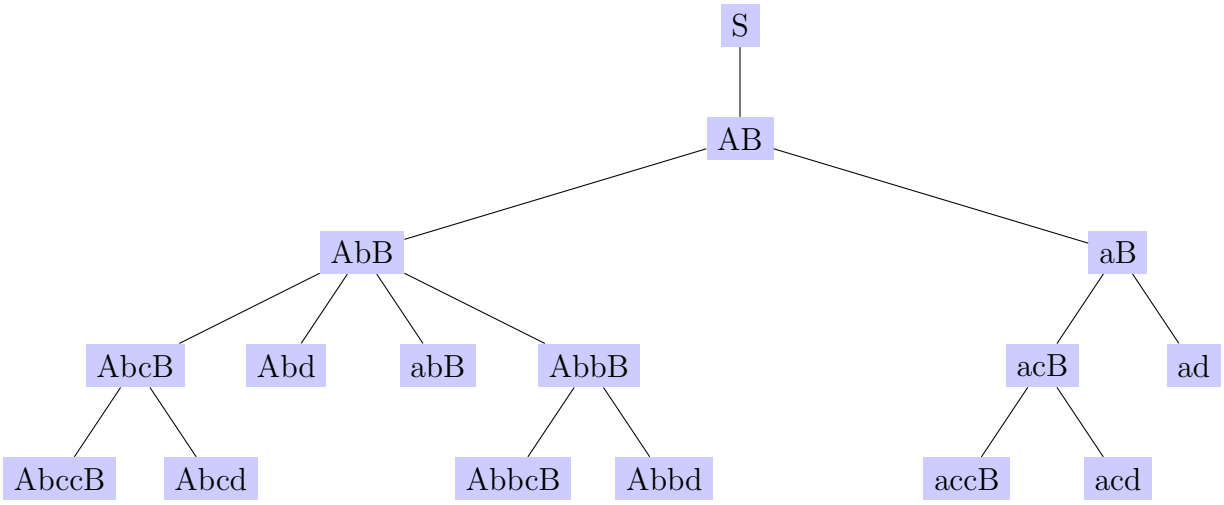
El siguiente paso que vamos a realizar será realizando transformaciones que no nos eliminen las variables. Vamos a empezar sustituyendo utilizando las producciones 1, 3 y 5 para seguidamente volver a aplicar las producciones. Teniendo:



Como modemos ver en el árbol de la página 2 hemos utilizado en el primero nivel la produccion 1 y en el segundo las producciones 3 y 5, que pertenecen a las producciones aplicables sobre A por último en el último nivel hemos utilizado el resto de producciones. Podemos ver como en todos

los casos siempre vamos a tener una ***a*** al principio. Por lo que modemos asegurar que la cadena será ***a ....***

Ahora vamos a continuar con el arbol, pero como ya sabemos que sucede con la variable A, ahora vamos a utilizar las producciones aplicables B teniendo:



Podemos ver ahora en el árbol que efectivamente todas las cadenas terminan en B o d y B finalmente es d por lo que podemos asegurar que la cadena será del estilo ***a ... b***. Por otro lado, podemos fijarnos que si vamos cambiando las variables por mas variables utilizando las producciones 2 y 3 llegamos a la conclusión de que en el interior de esta cadena tenemos tantas b y c como queramos, pudiendo ser 0 y siendo distinto numero de elementos. Por lo que finalmente:

El lenguaje que genera este gramática será {$a b^i c^j d$: $i,j \in \mathbb{N}$ }

1.3. Solución

Una vez visto el apartado anterior, vamos a crear las distintas reglas para obtener una gramática de tipo 3, para ello, tendremos las siguientes producciones.

1. **S → aB**
2. **B → bB**
3. **B → C**
4. **C → cC**
5. **C → d**

La produccion 1 la creamos claramente porque sabemos que la cadena debe ser del tipo ***a ....***

La produccion 2 nos permitirá añadir tantas *'b'* como queramos, teniendo ahora ab^i ***....***

La produccion 3 nos permitirá el cambio a la nueva variable C, para añadir tantas *'c'* como queramos a la cadena.

La produccion 4 como hemos dicho anteriormente, con esta produccion podremos añadir tantas *'c'* como queramos, teniendo ahora $ab^i c^j$ ***....***

Por último, la produccion 5 nos permitirá añadir la última *'d'* que nos falta al final, teniendo ahora $ab^i c^j d$.

2. Practica 2: Automata determinista JFLAP

2.1. Enunciado

Esta practica consiste en pasar un automata no determinista a un automata determinista utilizando JFLAP.

2.2. Proceso

En primer lugar, debemos tener instalado Java para poder ejecutar JFLAP en nuestro Linux. Por lo que debemos ejecutar:

```
$ sudo apt-get install default-jdk
```

Una vez tenemos instalado Java, ya podemos descargar JFLAP desde la web oficial[1] y podemos seguir los pasos indicados en en enlace[3] para ejecutar JFLAP.

2.3. Solución

Para la realizacion de la práctica haremos dos casos, primero cojeremos un caso 'simple' del tema y utilizaremos JFLAP para pasarlo a un automata no determinista. Seguidamente, en la segunda parte, utilizaremos un automata no determinista algo mas complejo para realizarle el mismo cambio utilizando de nuevo JFLAP.

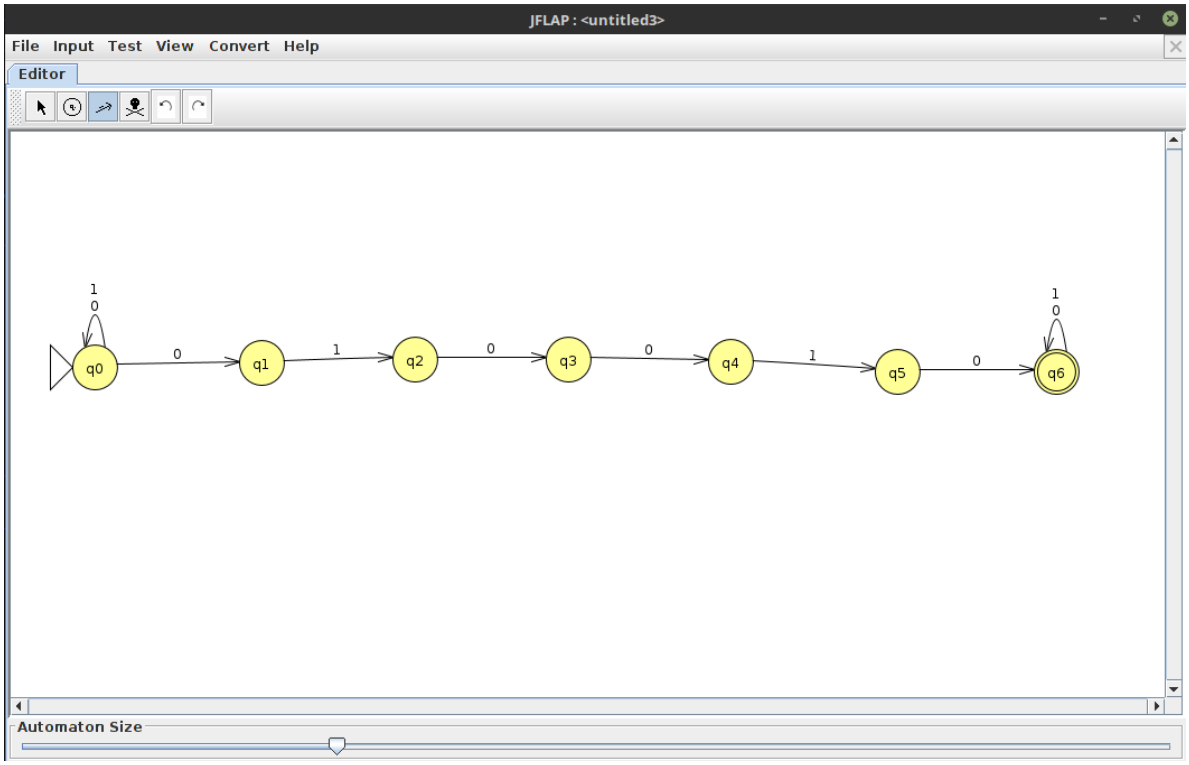
2.3.1. Caso 1: Automata mas simple

En este caso, vamos a utilizar el automata que admite cadenas que contengan la subcadena **010010**. Para ello, debemos abrir JFLAP y seguir los pasos indicados a continuación:

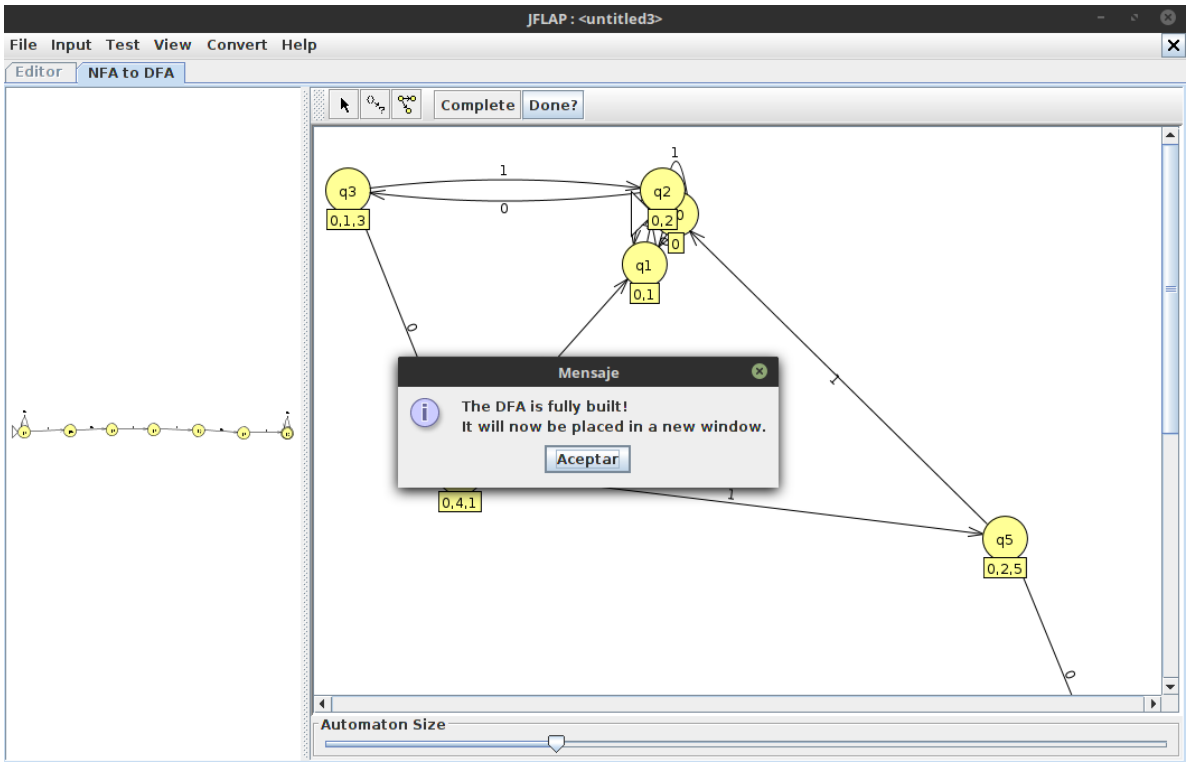
En primer lugar, posicionados en el directorio donde hemos descargado JFLAP, ejecutamos JFLAP.

```
$ java -jar JFLAP.jar
```

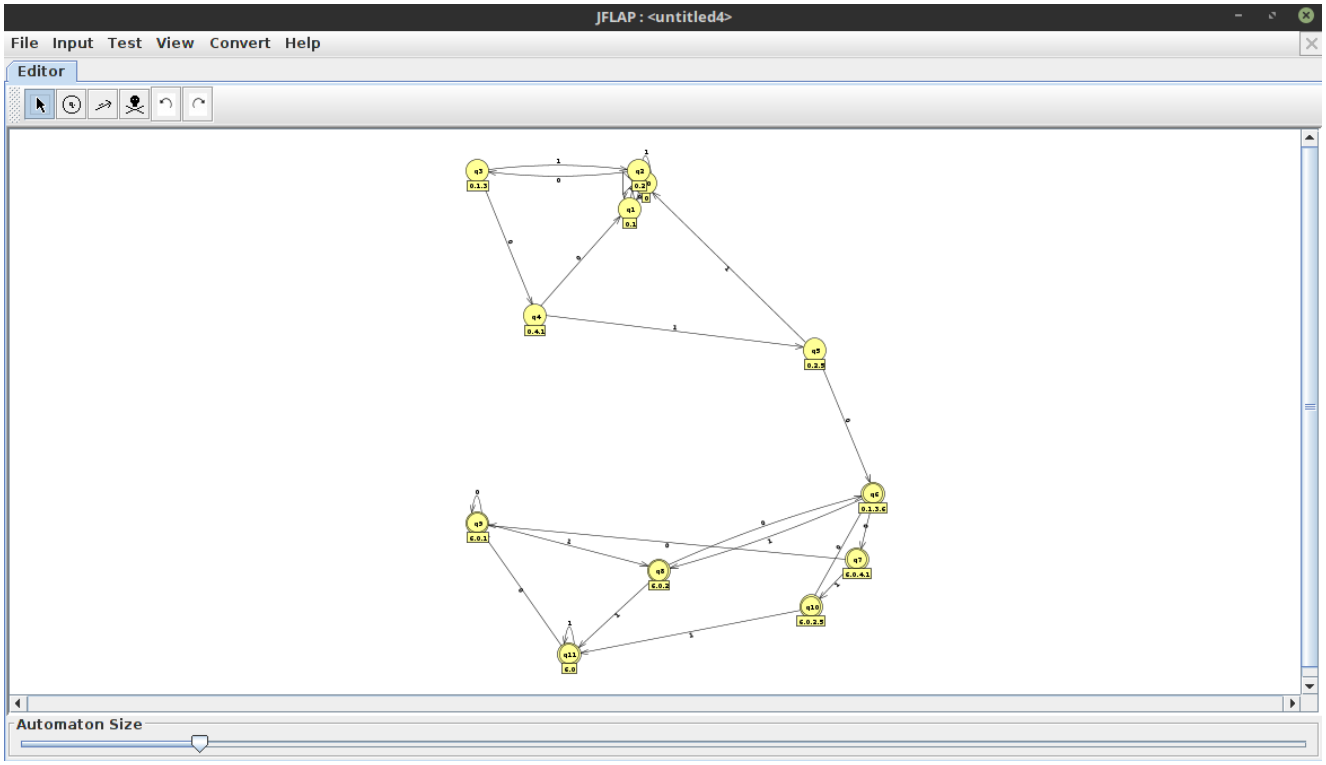
Una vez abierto, el siguiente paso es pinchar en ***Finite Automaton*** para poder dibujarlo:



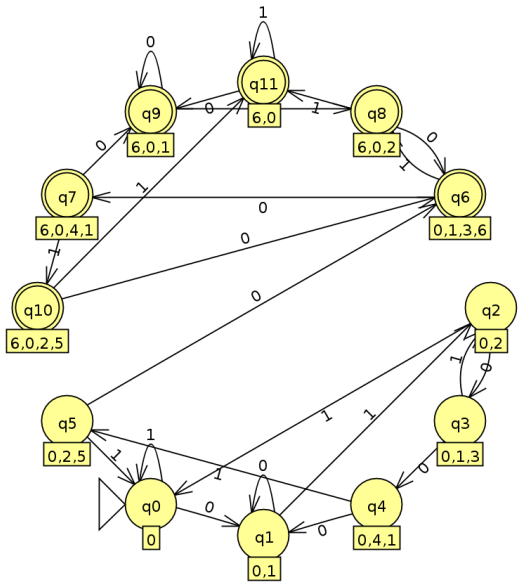
El siguiente paso es pasar este automata no determinista a otro determinista. Para ello, sólo tenemos que pulsar en ***Convert–Convert to DFA–Complete-Done?***



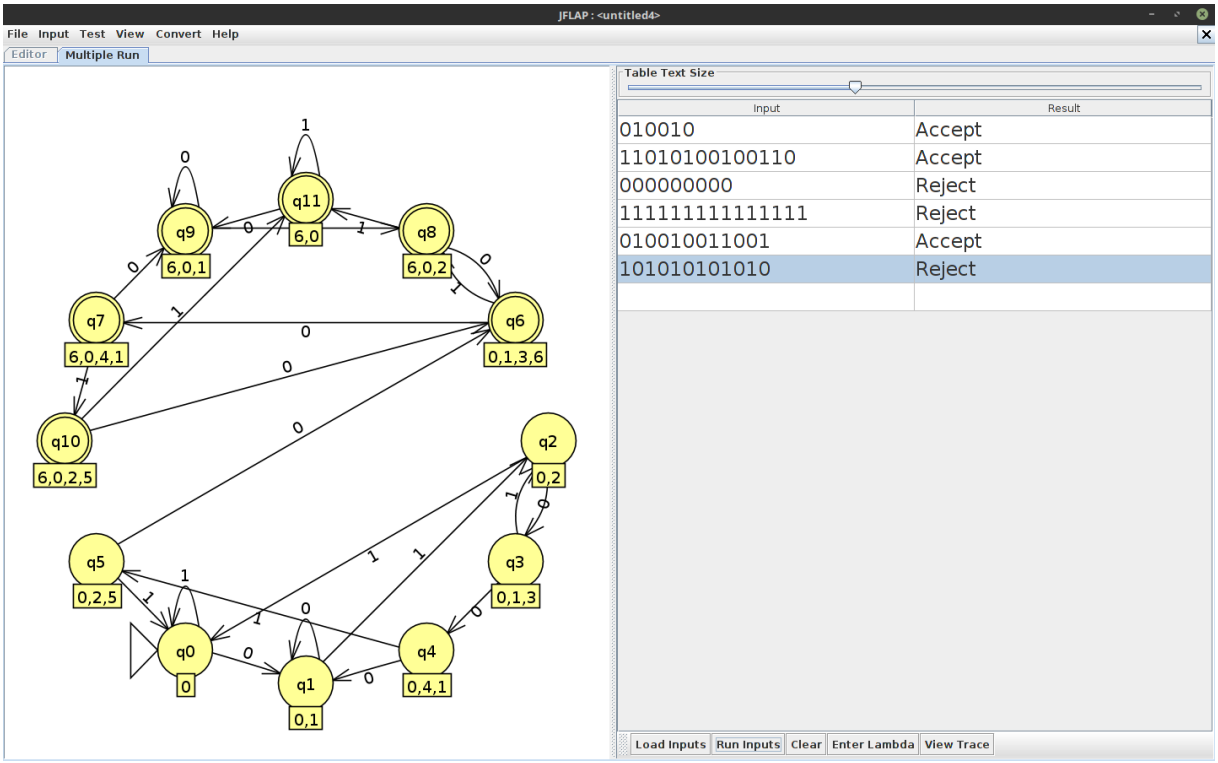
El programa nos muestra el automata de cualquier manera aleatoria y desordenada por lo que tenemos:



Lo único que podemos hacer para tener una mayor visibilidad es en ***View–Apply specific layout algorithm–[ELEGIR]*** para tener una visualizacion algo mejor.



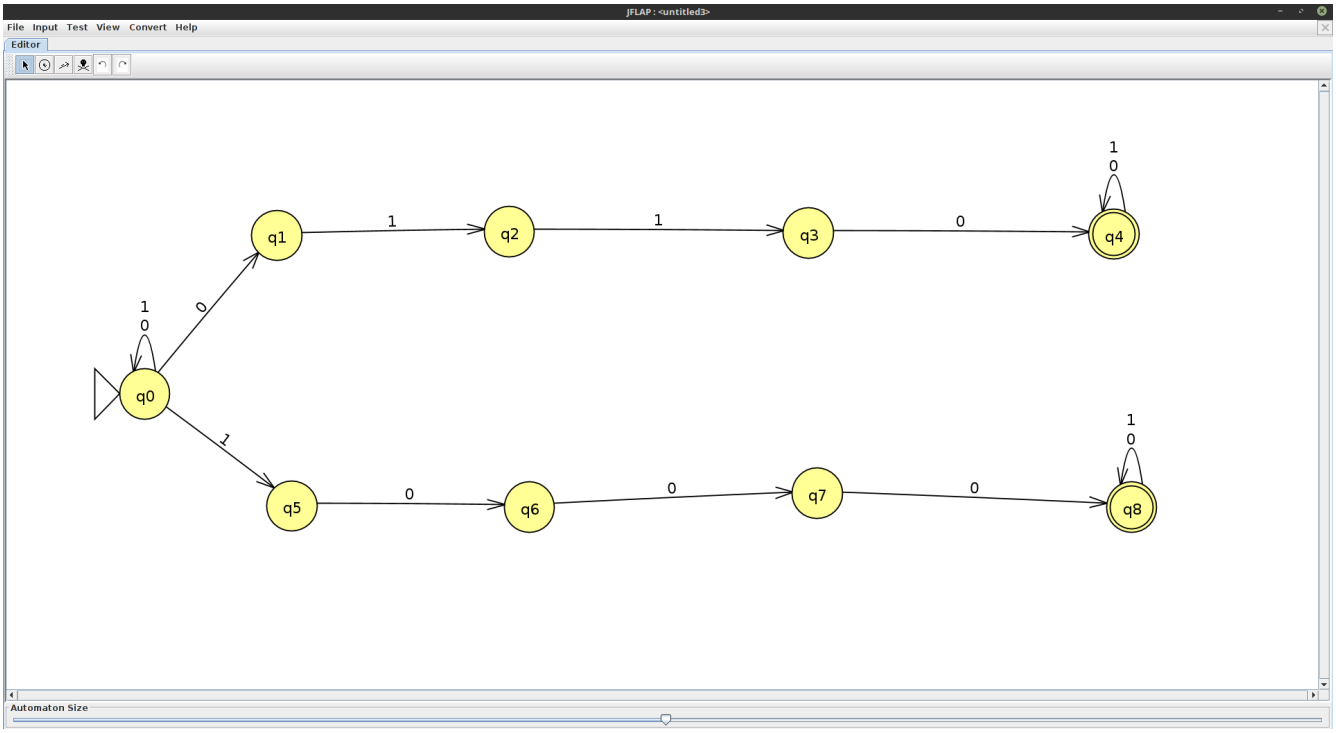
Por último, para comprobar que funciona correctamente, vamos a ***Input–Multiple Run*** para pasarle algunas cadenas y ver si efectivamente funciona.



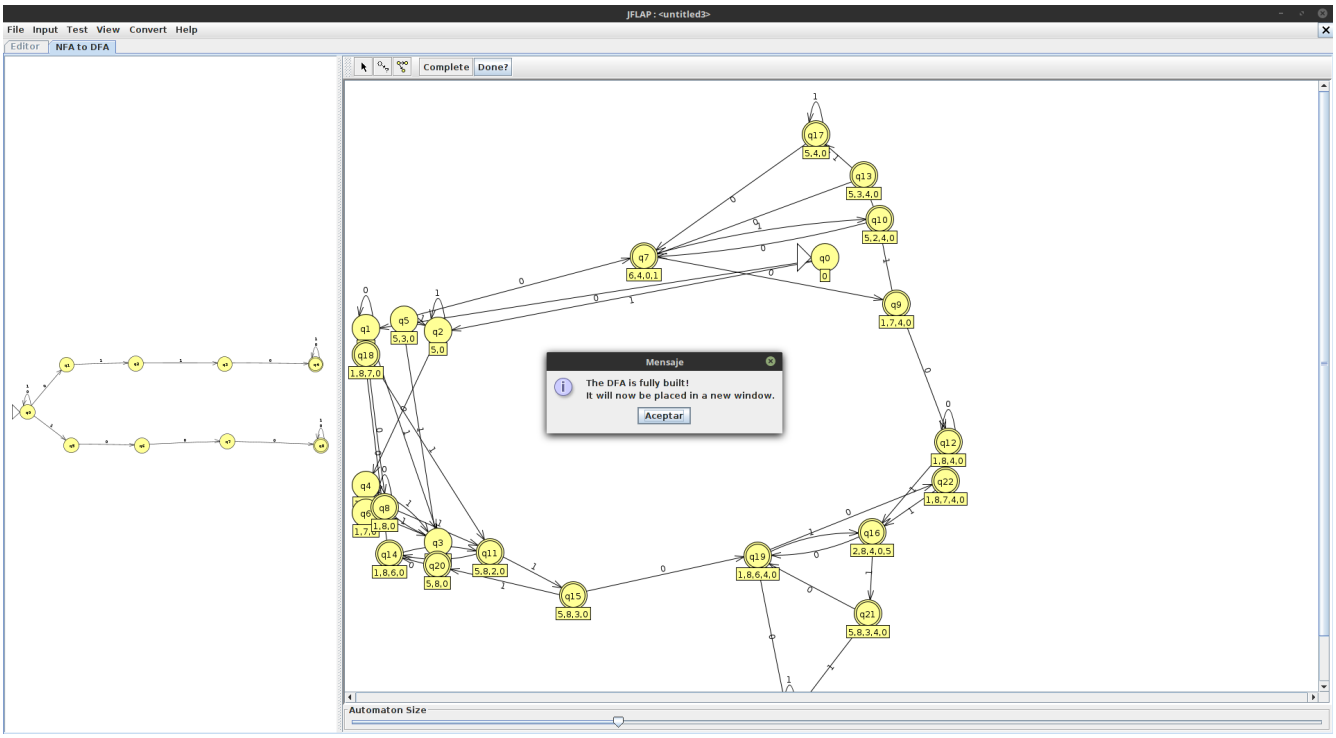
2.3.2. Caso 2: Autómata menos simple

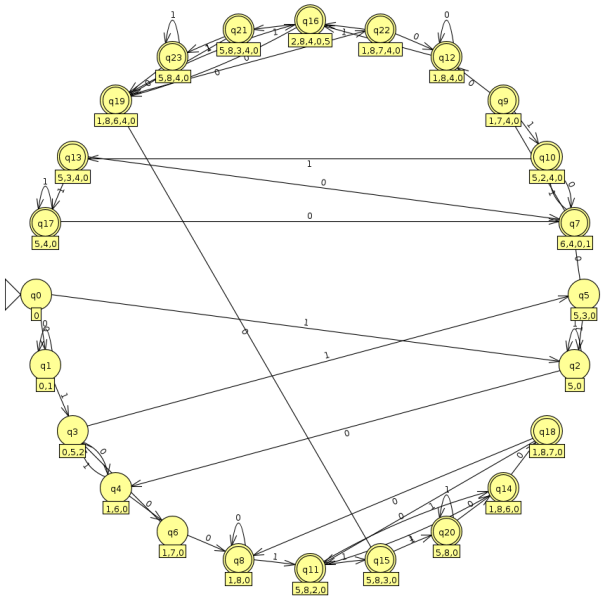
Vamos a hacer en este caso, la combinacion de lo anterior, con una nueva cadena, para que el mismo automata sea capaz de leer las dos cadenas.

En este caso, vamos a crear un autómata que sea capaz de reconocer las cadenas **0110** y **1000**.

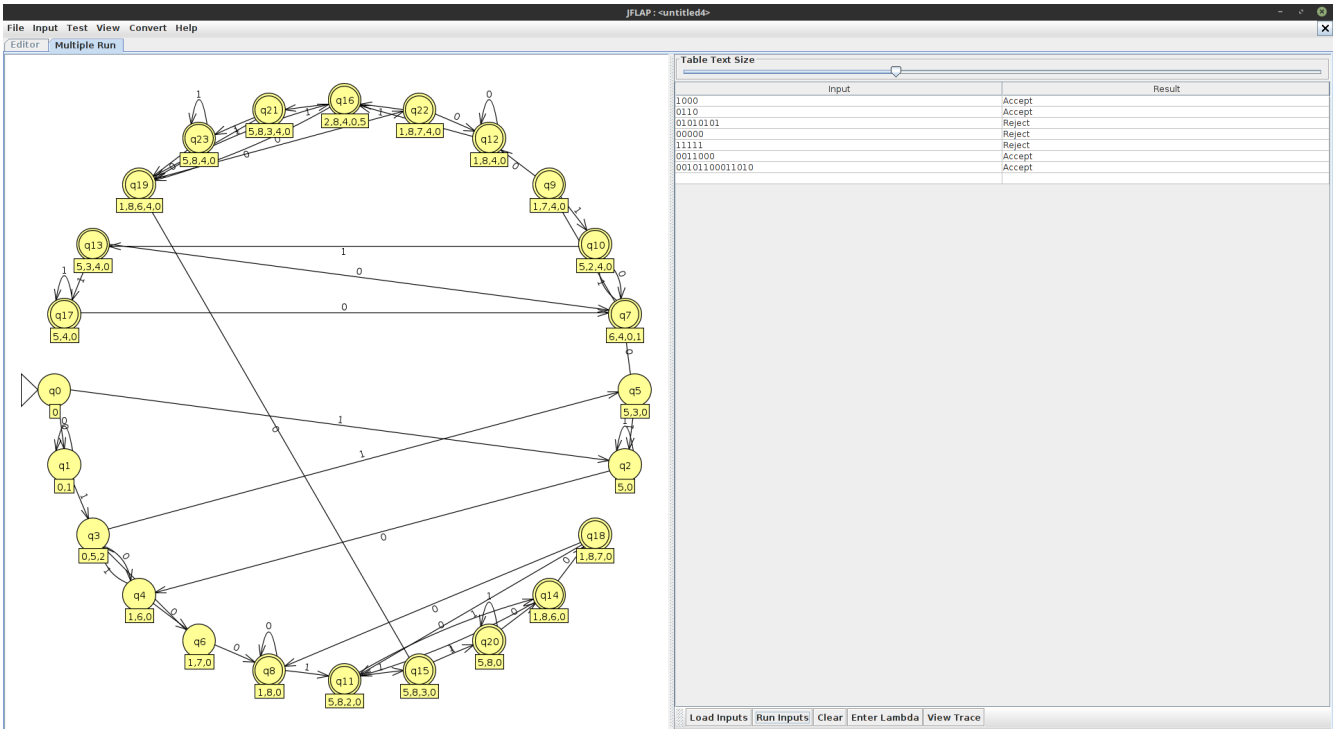


Vamos a pasarlo a AFD siguiendo los mismos pasos que hemos seguido en el apartado 2.3.1 de la página 5





Como podemos comprobar, la complejidad de este automata es mucho más alta que el anterior y eso que el cambio no ha sido tan drástico. Por último vamos a pasarle algunas cadenas para comprobar su funcionamiento.


JFLAP : <untitled4>
File Input Test View Convert Help
Editor
Multiple Run
q0 0
q1 0,1
q2 5,0
q3 0,5,2
q4 1,6,0
q5 5,3,0
q6 1,7,0
q7 6,4,0,1
q8 1,8,0
q9 1,7,4,0
q10 5,2,4,0
q11 5,8,2,0
q12 1,8,4,0
q13 5,3,4,0
q14 1,8,6,0
q15 5,8,3,0
q16 2,8,4,0,5
q17 5,4,0
q18 1,8,7,0
q19 1,8,6,4,0
q20 5,8,0
q21 5,8,3,4,0
q22 1,8,7,4,0
q23 5,8,4,0
Table Text Size
Input
Result
1000 Accept
0110 Accept
01010101 Reject
00000 Reject
11111 Reject
0011000 Accept
00101100011010 Accept
Load Inputs
Run Inputs
Clear
Enter Lambda
View Trace

3. Practica 3: Utilizacion de Lex

3.1. Enunciado

En esta practica, que trata de utilizar lex, cojiendo un programa hecho en lex[4] y compilandolo para poder comprobar su funcionamiento. En mi caso, voy a mostrar un programa que permite identificar los distintos elementos de un calculo matemático.

3.2. Proceso

Para esta practica sólo debemos tener instaldo gcc y lex para el correcto funcionamiento de la misma.

```
$ sudo apt-get install gcc flex
```

3.3. Solución

Lo primero de todo será crear nuestro fichero lex, que en mi caso es el siguiente[2]:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%option yylineno
%option noyywrap
ID      [A-Za-z_][A-Za-z_0-9]*
INT     -?[1-9][0-9]*
OP      [-+*/^=]
%%
 /* Pintar delimitadores. */
[(]             {printf("(Parentesis-izquierdo %u)\n", yylineno);}
[)]             {printf("(Parentesis-derecho %u)\n", yylineno);}
[;]             {printf("(Punto-y-coma %u)\n", yylineno);}
 /* Pintar identificadores , enteros y operadores. */
{INT}           {printf("(int %s %u)\n",yytext, yylineno);}
{ID}            {printf("(id \"%s\" %u)\n", yytext, yylineno);}
{OP}            {printf("(op \"%s\" %u)\n", yytext, yylineno);}
 /* Ignorar comentarios y espacios en blanco. */
#[^\n]*         {}
[ \t\r\n]       {}
<<EOF>>         {printf("(eof %u)\n", yylineno); return 0;}
%%
int main(int argc, char* argv[]) {
  yylex() ;
  return EXIT_SUCCESS ;
}
```

Vamos a explicar un poco como funciona este fichero lex:

En primer caso, en las lineas **1-4** lo que hacemos es incluir las librerias necesarias en C que necesitamos, seguidamente en las líneas **5-6** lo que añadimos son dos opciones:

- option yylineno → Hace que el numero de línea actual esté disponible en la variable C global yylineno.
- option noyywrap → Que evita que el lexer generado invoque la función yywrap proporcionada por el usuario cuando llega al final del archivo actual.

En las líenas **7-9** lo que indicamos son el nombre de las variables que vamos a utilizar siendo:

- ID → Posibles nombres para las variables.
- INT → Numeros enteros.
- OP → Operadores para el calculo

Una vez declaradas estas variables, en las líneas **12-14** comprobamos tambien otros posibles elementos como:

- '(' → Parentesis que abre, ya sea para dar prioridad o para definir valores de funciones.
- ')' → Parentesis que cierra.
- ';' → Punto y coma (como final de línea).

Despues, en las líneas **16-18** es lo que se realiza cuando se encuentra cualquiera de las variables declaradas anteriormente. Simplemente mostramos el valor de las mismas como veremos en el ejemplo gráfico.

En las líneas **20-21** Lo que hacemos simplemente es descartar las líneas que sean comentarios y los espacios en blanco.

En la línea **23** mostramos el final de fichero.

Por último, en las líneas **24-26** es el propio main() que se ejecuta cuando ejecutamos el programa lex.

Una vez explicado esto, pasamos a compilar el programa y a ejecutarlo, pero antes voy a mostar el fichero de entrada que he creado para este programa lex.

```
1 3 + 5
  x = 3
3 4 + x
  f(x) = x^2
5 f(20) + 10
```

Los pasos a realizar serían los siguientes:

```
$ lex fichero.l
$ gcc lex.yy.c -o ficheroexe -ll
$ cat entrada | ./ficheroexe > salida
```

Como resultado, obtendremos lo siguiente:

```
1 (int 3 1)
  (op "+" 1)
3 (int 5 1)
  (id "x" 2)
5 (op "=" 2)
```

```
(int 3 2)
(int 4 3)
(op "+" 3)
(id "x" 3)
(id "f" 4)
(Parentesis-izquierdo 4)
(id "x" 4)
(Parentesis-derecho 4)
(op "=" 4)
(id "x" 4)
(op "^" 4)
(int 2 4)
(id "f" 5)
(Parentesis-izquierdo 5)
(int 20 5)
(Parentesis-derecho 5)
(op "+" 5)
(int 10 5)
(eof 6)
```

En el que la estructura de salida es (TIPO ELEMENTO NUMERO_OPERACION). Por lo que por ejemplo en la primera operacion (3+5) tenemos como resultado:

- (int 3 1)
- (op "+" 1)
- (int 5 1)

Donde el primero indica que tenemos un **entero** que es el numero **3** y que pertenece al **numero de operacion 1** de todas las operaciones que tenemos. Donde el segundo indica que tenemos un **operador** que es el + y que pertenece al **numero de operacion 1** de todas las operaciones que tenemos. Y donde el tercero indica que tenemos otro **entero** que es el numero **5** y que pertenece al **numero de operacion 1** de todas las operaciones que tenemos.

4. Practica 4: Forma normal de Chomsky y algoritmo de Cocke-Younger-Kasami

4.1. Enunciado

En esta practica, en primer lugar voy a pasar una gramatica libre del contexto a la forma normal de Chomsky, para seguidamente aplicarle el algoritmo de Cocke-Younger-Kasami a esa gramática en forma normal de Chomsky. Siendo la gramática a utilizar la siguiente:

1. **S** $\rightarrow$ **bA**
2. **S** $\rightarrow$ **aB**
3. **A** $\rightarrow$ **bAA**
4. **A** $\rightarrow$ **AS**
5. **A** $\rightarrow$ **a**
6. **B** $\rightarrow$ **aBB**
7. **B** $\rightarrow$ **bS**
8. **B** $\rightarrow$ **b**

4.2. Solucion

4.2.1. Forma normal de Chomsky

El primer paso que se realiza para pasar a la forma normal de Chomsky es **eliminar terminales en producciones que no sean** $\boldsymbol{A \rightarrow a}$ por lo que:

1. ~~S $\rightarrow$ bA~~:
   - S $\rightarrow C_b$A
2. ~~S $\rightarrow$ aB~~:
   - S $\rightarrow C_a$B
3. ~~A $\rightarrow$ bAA~~:
   - A $\rightarrow C_b$AA
4. ~~B $\rightarrow$ aBB~~:
   - B $\rightarrow C_a$BB
5. ~~B $\rightarrow$ bS~~:
   - B $\rightarrow C_b$S

Por lo que finalmente, realizado el primer paso del algoritmo, tenemos las siguiente producciones:

1. **S** $\rightarrow C_b$**A**
2. **S** $\rightarrow C_a$**B**
3. **A** $\rightarrow C_b$**AA**
4. **A** $\rightarrow$ **AS**
5. **A** $\rightarrow$ **a**
6. **B** $\rightarrow C_a$**BB**

7. **B** $\rightarrow C_b$**S**
8. **B** $\rightarrow$ **b**
9. $C_a \rightarrow$ **a**
10. $C_b \rightarrow$ **b**

Una vez tenemos esto, ya podemos aplicar el segundo paso y último del algoritmo, que trata de **eliminar terminales en producciones que no sean $A \rightarrow AA$ por lo que:**

1. ~~A $\rightarrow C_b$AA~~:
   - A $\rightarrow C_b D_1$
   - $D_1 \rightarrow$ AA
2. ~~B $\rightarrow C_a$BB~~:
   - B $\rightarrow C_a E_1$
   - $E_1 \rightarrow$ BB

Por lo que finalmente, realizado el último paso del algoritmo, tenemos las siguiente producciones y por lo tanto, en forma normal de Chomsky:

Producciones finales

1. **S** $\rightarrow C_b$**A**
2. **S** $\rightarrow C_a$**B**
3. **A** $\rightarrow$ **AS**
4. **A** $\rightarrow$ **a**
5. **B** $\rightarrow C_b$**S**
6. **B** $\rightarrow$ **b**
7. $C_a \rightarrow$ **a**
8. $C_b \rightarrow$ **b**
9. **A** $\rightarrow C_b D_1$
10. $D_1 \rightarrow$ **AA**
11. **B** $\rightarrow C_a E_1$
12. $E_1 \rightarrow$ **BB**

4.2.2. Algoritmo de Cocke-Younger-Kasami

Ahora, vamos a pasarle el algoritmo CYK a las producciones creadas anteriormente en forma normal de Chomsky con la siguiente cadena:

- **abba**:

  En primer paso, rellenamos la primera fila, ponniendo unicamente las producciones que permiten crear directamente el valor final indicado en la columna.

a	**b**	**b**	**a**
A C_a	B C_b	B C_b	A C_a

  Ahora, para rellenar la segunda fila, vamos comprobando si existe alguna combinacion de las anteriores que nos permita crear la cadena con 2 elementos de longitud.

a	**b**	**b**	**a**
A C_a	B C_b	B C_b	A C_a
S			

La conmbinación $C_a\boldsymbol{B}$, la podemos producir a través de S y por eso la añadimos.

a	**b**	**b**	**a**
A C_a	B C_b	B C_b	A C_a
S	E_1		

La conmbinación $\boldsymbol{BB}$, la podemos producir a través de E_1 y por eso la añadimos.

a	**b**	**b**	**a**
A C_a	B C_b	B C_b	A C_a
S	E_1	S	

La conmbinación $C_b\boldsymbol{A}$, la podemos producir a través de S y por eso la añadimos.

a	**b**	**b**	**a**
A C_a	B C_b	B C_b	A C_a
S	E_1	S	
B			

La conmbinación C_aE_1, la podemos producir a través de B y por eso la añadimos. No pongo el procedimiento de las siguientes comprobaciones para la misma celda porque no existen posibles combinaciones que añadan mas producciones.

a	**b**	**b**	**a**
A C_a	B C_b	B C_b	A C_a
S	E_1	S	
B	B		

La conmbinación $C_b\boldsymbol{S}$, la podemos producir a través de B y por eso la añadimos. No pongo el procedimiento de las siguientes comprobaciones para la misma celda porque no existen posibles combinaciones que añadan mas producciones.

a	**b**	**b**	**a**
A C_a	B C_b	B C_b	A C_a
S	E_1	S	
B	B		
S			

La conmbinación $C_a\boldsymbol{B}$, la podemos producir a través de S y por eso la añadimos. No pongo el procedimiento de las siguientes comprobaciones para la misma celda porque no existen

posibles combinaciones que añadan mas producciones.

Por lo que finalmente podemos concluir que la cadena **abba** pertenece al lenguaje ya que tenemos en la última casilla la producción **S** que es la producción de inicio.

Referencias

[1] Web oficial JFLAP, jflap.org, `http://www.jflap.org/`, Accedido el 8 de enero de 2018.

[2] Matt Might, Ejemplo fichero lex, leyendo operaciones, `http://matt.might.net/articles/standalone-lexers-with-lex/` Accedido el 8 de enero de 2018

[3] Web oficial JFLAP, pasos para la ejecucion, jflap.org, `http://www.jflap.org/jflaptmp/toRun.html` Accedido el 8 de enero de 2018

[4] M. E. Lesk and E. Schmidt , Lex - A Lexical Analyzer Generator, dinosaur.compilertools.net, `http://dinosaur.compilertools.net/lex/` Accedido el 8 de enero de 2018