Prácticas de MC

Carlos Enriquez López - 77392884T - Grupo B

Práctica 1:

Determinar si la gramática G = ({S,A,B},{a,b,c}, P,S) genera un lenguaje del tipo 3 (lenguaje regular). P es el conjunto de reglas de producción siguiente:

S -> AB A -> Ab A -> a B -> cB B -> d

Solución:

Esta gramática genera el lenguaje {ab^ic^jd : i,j € N}

Dada cualquier cadena del lenguaje podemos generar, mediante reglas de producción, una cadena de símbolos terminales del tipo ab^ic^jd. Esto quiere decir que la cadena final tiene tantas *b* y *c* como se quiera en esa posición pero solo una *a* al principio y una *d* al final. Ejemplos: *abbbccd, abcd, abcccd.*

La gramática dada no es regular (como podemos comprobar por la producción S -> AB ya que genera más de una variable), por lo tanto, en principio, no genera lenguajes regulares. Esto quiere decir que el lenguaje generado no debería ser de tipo 3

Sin embargo, existe otra gramática regular que genera el mismo lenguaje:

S -> aB B -> bB B -> C C -> cC C –> d

Como esta gramática si es regular y genera el lenguaje inicial llegamos a la conclusión de que el lenguaje generado si es de tipo 3 (regular) y por lo tanto la primera gramática si puede generar un lenguaje tipo 3.

Práctica 2:

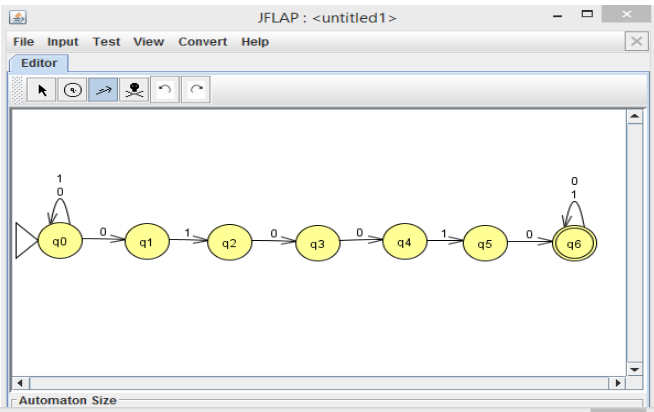
Usando JFLAP:

Transformar un autómata no determinístico a determinístico, probar algunas cadenas e imprimir el resultado. Repetir el proceso pero con un autómata con transiciones nulas.

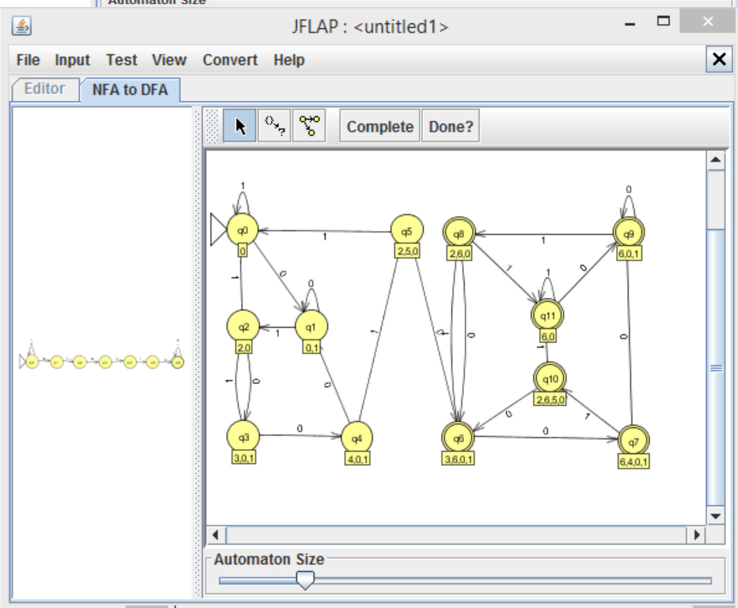
El autómata usado solo acepta cadenas que contengan la sucesión 010010 y rechaza el resto.

Solución:

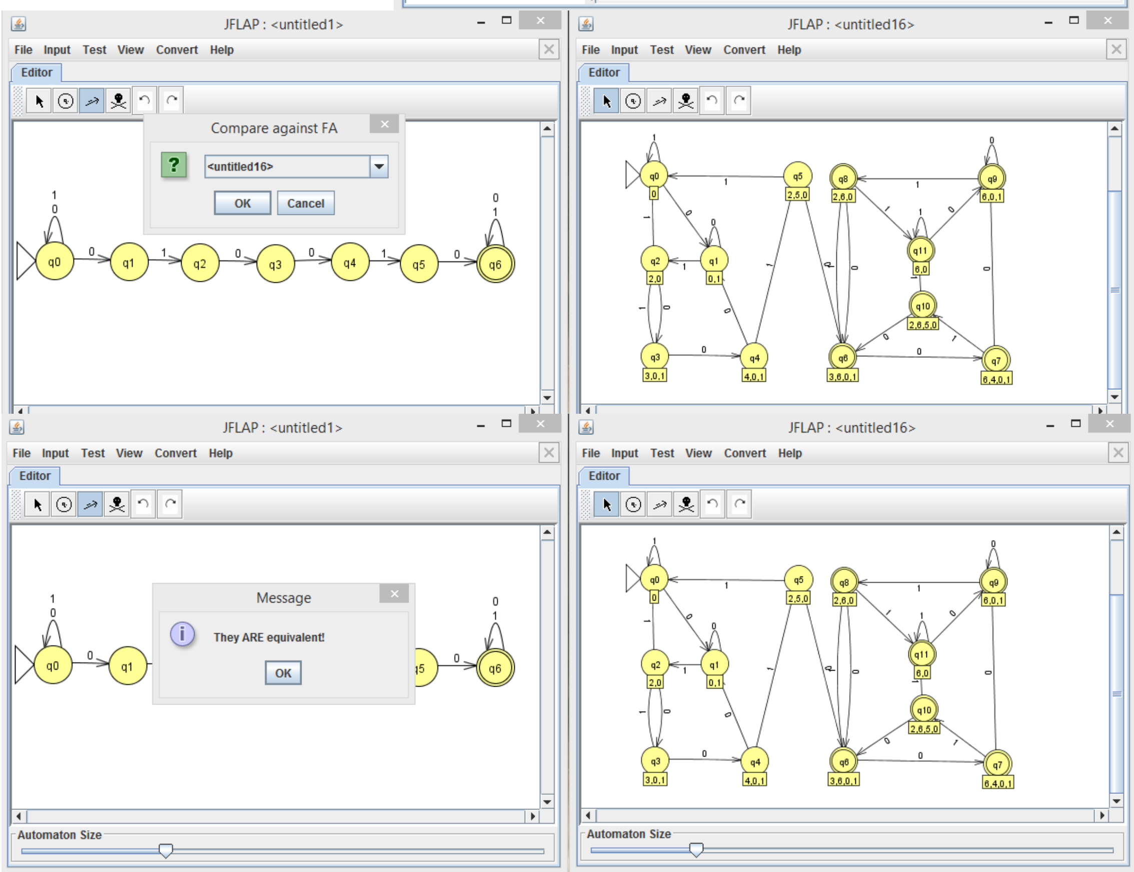
Diseñamos el NFA (autómata finito no determinístico) que acepta las cadenas que queremos.



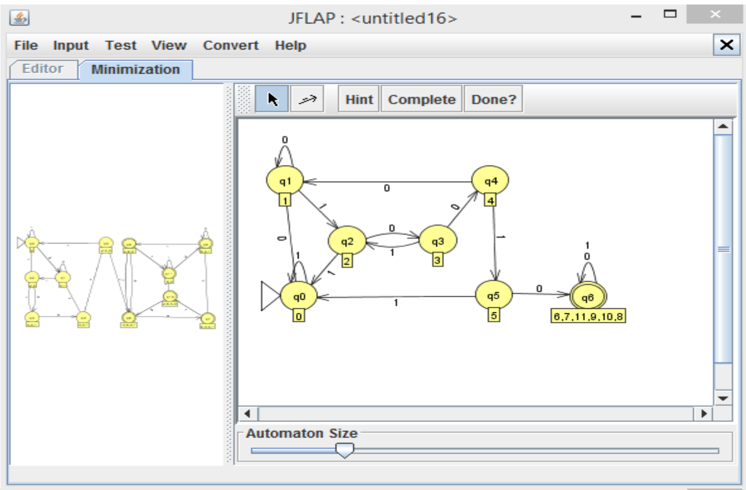
Con JFLAP pasamos el NFA anterior a uno DFA (autómata finito determinístico).



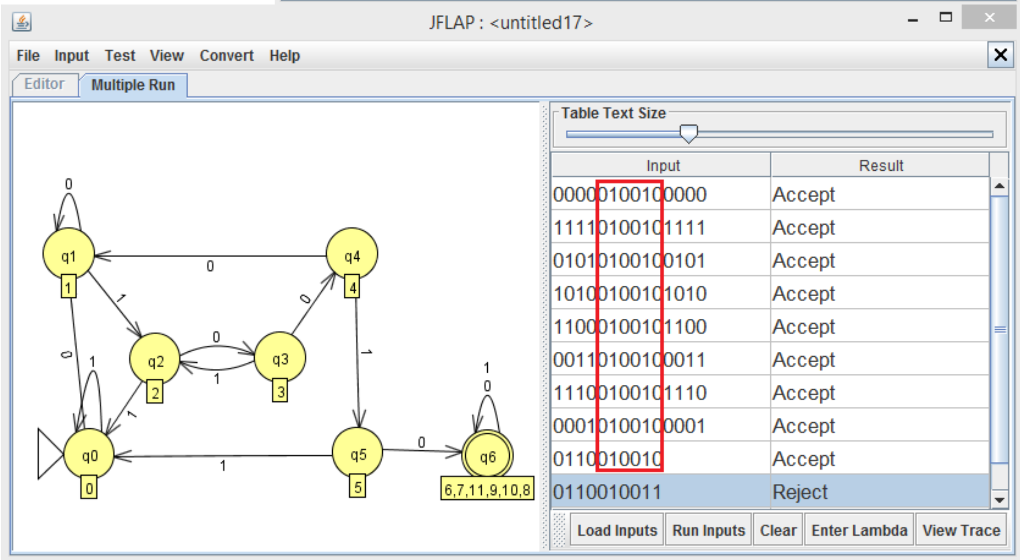
Los comparamos también con JFLAP y vemos que son equivalentes.



Minimizamos el DFA obtenido con JFLAP.



Por último probamos algunas cadenas en el autómata resultante.



Ahora hacemos lo mismo pero para un autómata que acepte transiciones nulas. En este caso queremos que acepte las cadenas que contengan cualquier subcadena que contenga la sucesión 1000 y bien la sucesión 0110 o ambas.

Realizamos el mismo proceso.

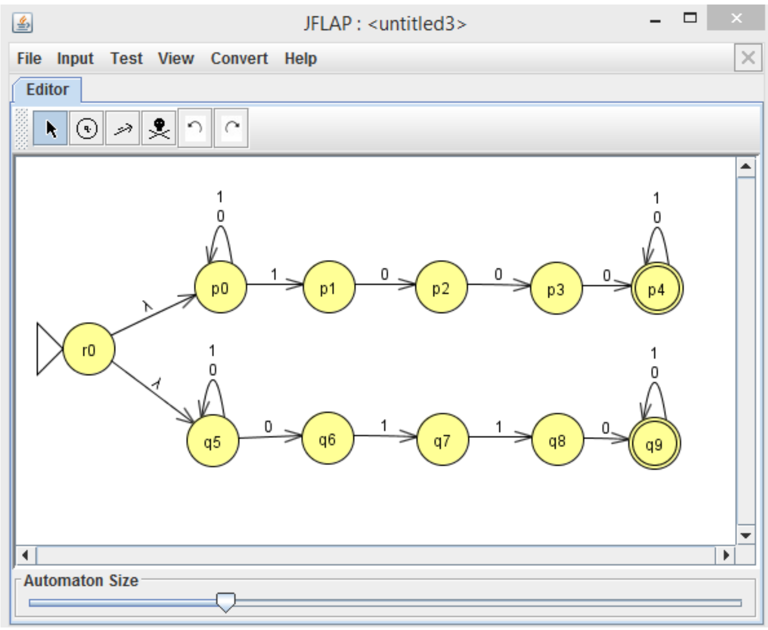
Creamos el NFA con transiciones nulas.

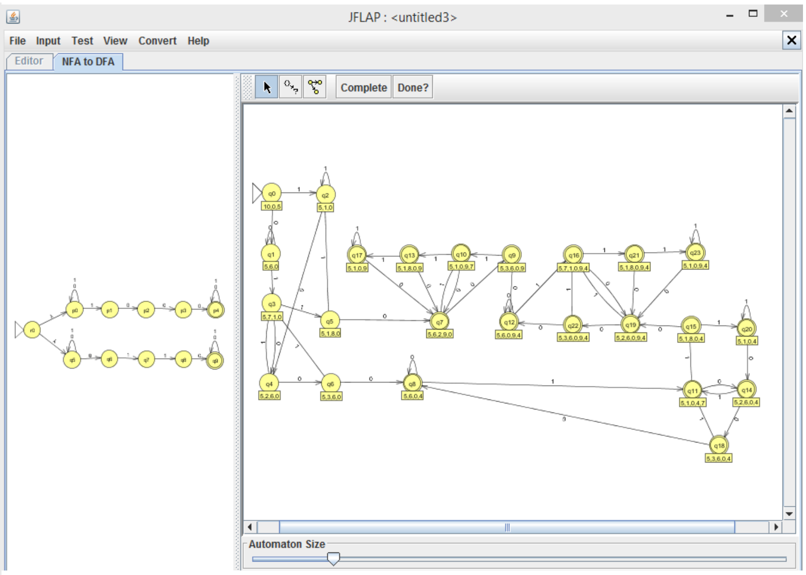
Con JFLAP pasamos el NFA anterior a uno DFA

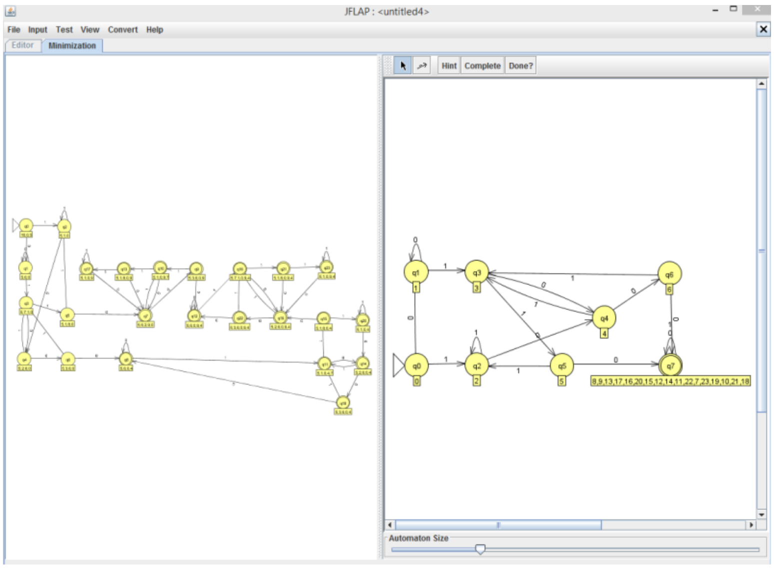
Con JFLAP pasamos el NFA anterior a uno DFA

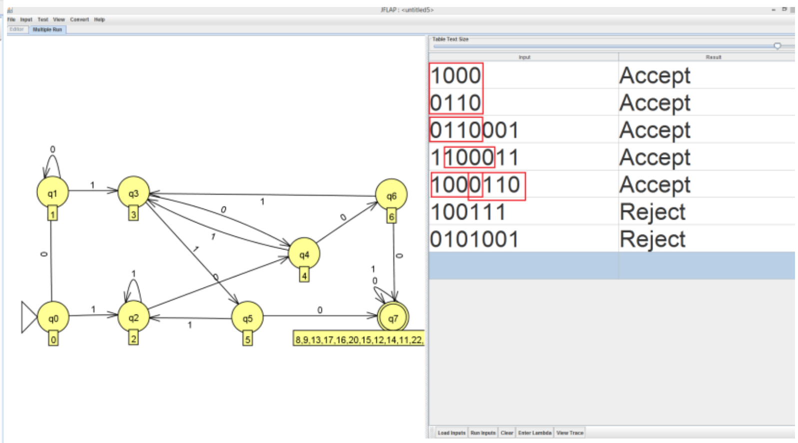
Los comparamos, minimizamos y probamos algunas cadenas.

Creamos un autómata que es combinación de dos autómatas que aceptan las cadenas que queremos.









Como podemos comprobar el autómata resultante acepta las cadenas que aceptaban los dos autómatas con los que hemos creado este.

Práctica 3:

En esta práctica utilizaremos el programa Lex para localizar expresiones regulares con acciones asociadas.

Solución:

Para ello haremos uso de un fichero como el siguiente:

```
car       [a-zA-Z]
digito    [0-9]
signo     (\-|\+)
suc       ({digito}+)
enter     ({signo}?{suc})
real1     ({enter}\.{digito}*)
real2     ({signo}?\.{suc})
   int    ent=0,   real=0,  ident=0, sumaent=0;

%%

   int i;
{enter}                    {ent++; sscanf(yytext,"%d",&i); sumaent+=i; printf("Numero
entero %s\n",yytext);}
({real1}|{real2})          {real++; printf("Num. real %s\n",yytext);}
{car}({car}|{digito})*     {ident++; printf("Var. ident. %s\n",yytext);}
.|\n                       {;}

%%

yywrap()
   {printf("Numero de Enteros %d, reales %d, ident %d, Suma Enteros %
d",ent,real,ident,sumaent); return 1;}
```

Este especifica varias cosas:

Una serie de declaraciones de variables como las variables de las expresiones que buscaremos y otras variables que usaremos con otros motivos.

Una serie de reglas a realizar cuando se detecten una serie de variables especificadas en un orden determinado.

Por últimos algunos procedimientos que queremos que se realicen con las variables.

Una vez creado el fichero tenemos que pasárselo al programa Lex. Lo podemos hacer mediante la línea de comandos de Ubuntu. (En este caso el fichero se llama lex).

```
carlos@carlos-HP:~/Escritorio$ lex lex
```

Este proceso da como resultado un fichero escrito en C que tendremos que compilar añadiéndole las librerías correspondientes. El fichero resultante será el siguiente:

lex.yy.c (~/Escritorio) - gedit

Abrir

```c

#line 3 "lex.yy.c"

#define  YY_INT_ALIGNED short int

/* A lexical scanner generated by flex */

#define FLEX_SCANNER
#define YY_FLEX_MAJOR_VERSION 2
#define YY_FLEX_MINOR_VERSION 6
#define YY_FLEX_SUBMINOR_VERSION 0
#if YY_FLEX_SUBMINOR_VERSION > 0
#define FLEX_BETA
#endif

/* First, we deal with  platform-specific or compiler-specific issues. */

/* begin standard C headers. */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>

/* end standard C headers. */

/* flex integer type definitions */

#ifndef FLEXINT_H
#define FLEXINT_H

/* C99 systems have <inttypes.h>. Non-C99 systems may or may not. */

#if defined (__STDC_VERSION__) && __STDC_VERSION__ >= 199901L

/* C99 says to define __STDC_LIMIT_MACROS before including stdint.h,
 * if you want the limit (max/min) macros for int types.
 */
#ifndef __STDC_LIMIT_MACROS
#define __STDC_LIMIT_MACROS 1
#endif

#include <inttypes.h>
typedef int8_t flex_int8_t;
typedef uint8_t flex_uint8_t;
typedef int16_t flex_int16_t;
typedef uint16_t flex_uint16_t;
typedef int32_t flex_int32_t;
typedef uint32_t flex_uint32_t;
#else
typedef signed char flex_int8_t;
typedef short int flex_int16_t;
typedef int flex_int32_t;
typedef unsigned char flex_uint8_t;
typedef unsigned short int flex_uint16_t;
typedef unsigned int flex_uint32_t;

```

Ahora compilamos el fichero C de esta manera lo cual nos generará un programa C que deberemos usar con un texto como argumento.

```
carlos@carlos-HP:~/Escritorio$ gcc lex.yy.c -o prog -lfl
lex:21:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
  {printf("Numero de Enteros%d, reales%d, ident%d, Suma Enteros%d",ent,real,iden
 ^
```

En texto válido seria por ejemplo el siguiente:

```
Texto de ejemplo:
1 -13 +99 2.1 -3.567 12.23

```

Al pasar el fichero del texto por el programa obtenido este produciría el resultado que se muestra en la siguiente imagen:

```
Var. ident. Texto
Var. ident. de
Var. ident. ejemplo
Numero entero 1
Numero entero -13
Numero entero +99
Num. real 2.1
Num. real -3.567
Num. real 12.23
Numero de Enteros 3, reales 3, ident 3, Suma Enteros 87
```

Claramente el programa reconoce las cadenas que se le indican que en este caso son números y nos dice de qué tipo son además de contar la cantidad de cada tipo de número y la suma de los enteros.

Práctica 4:

Esta práctica consistirá en pasar una gramática libre de contexto a la forma normal de Chomsky y con el algoritmo de Cocke-Younger-Kasami ver si una serie de palabras pertenecen a dicha gramática.

Solución:

Utilizaremos la siguiente gramática libre de contexto (sabemos que es así porque solo tiene una variable a la izquierda de todas las producciones):

S -> bA	S -> aB	A -> bAA	A -> AS
A -> a	B -> aBB	B -> bS	B -> b

Para usar el algoritmo de Cocke-Younger-Kasami para ver si una palabra pertenece a dicha gramática primero tenemos que tener la gramática en forma normal de Chomsky.

Transformamos la gramática a la forma normal de Chomsky:

S -> bA	S -> aB	A -> bAA	A -> AS
A -> a	B -> aBB	B -> bS	B -> b

$\downarrow$

S -> C_bA	S -> C_aB	A -> C_bAA	A -> AS
A -> a	B -> C_aBB	B -> C_bS	B -> b
C_a -> a	C_b -> b		

$\downarrow$

S -> C_bA S -> C_aB A -> C_bAA A -> AS

A -> a B -> C_aBB B -> C_bS B -> b

C_a -> a C_b -> b A -> C_bD_1 D_1 -> AA

S -> C_bA S -> C_aB A -> AS

A -> a B -> C_aBB B -> C_bS B -> b

C_a -> a C_b -> b A -> C_bD_1 D_1 -> AA

B -> C_aE_1 E_1 -> BB

S -> C_bA S -> C_aB A -> AS A -> a

B -> C_bS B -> b C_a -> a C_b -> b

A -> C_bD_1 D_1 -> AA B -> C_aE_1 E_1 -> BB

Ahora que hemos acabado tenemos la gramática en la forma normal de Chomsky y podemos aplicar el algoritmo de Cocke-Younger-Kasami con una serie de palabras para ver si estas pertenecen a dicha gramática. Probaremos el algoritmo con las palabras *abba* y *abbb.*

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

Como vemos, en la primera palabra el algoritmo llega al símbolo generador S por lo tanto la palabra se puede formar a través de la gramática (*S – aB – abS – abbA – abba*), sin embargo en el segundo caso la gramática no puede generar la palabra ya que el algoritmo no nos lleva al símbolo generador S.