**ETSIIT – Universidad de Granada (UGR)**

**Grado en Ing. Informática – 3ºA – Modelos de Computación**

# PRÁCTICAS. Subgrupo B2

# ÍNDICE

**ETSIIT – Universidad de Granada (UGR)**

**Grado en Ing. Informática – 3ºA – Modelos de Computación**

# 1. PRÁCTICA 1.- Optimizando gramáticas

### 1.1 Apartado a) Determinar si la gramática G = ({S,A,B},{a,b, c,d},P,S) donde P es el conjunto de reglas de producción:

***S → AB*** ***A → Ab*** ***A → a*** ***B → cB*** ***B → d***

**genera un lenguaje de tipo 3.**

Sabemos que esta gramática genera el siguiente lenguaje: ***{$ab^ic^jd$ : 0<=i<=j i,j ∈ N}***

Para llegar aquí, hemos seguido los siguientes pasos:

S→AB, S→AbB, S→$Ab^2B$, … S→AbB, S→$ab^iB$,
S→$ab^icB$, S→$ab^ic^2B$, ... S→$ab^ic^jB$, **S→$ab^ic^jd$**

No es una lenguaje regular, ya que nos encontramos dos variables en la parte derecha, y las variables deberían aparecer en uno de los lados.

Por tanto, llegamos a la conclusión de que es un lenguaje de **tipo 2**, es decir, ***Lenguajes Independientes del Contexto.*** Este tipo de lenguaje es menos restrictivo y nos permite tener todo tipo de combinaciones.

### 1.2 Apartado b) ¿Se puede buscar una gramática lineal para crear un lenguaje de tipo 3?

Si, se puede buscar este tipo de gramática un ejemplo sería:

S → aB B → bB B → C C → cC C → d

Al ser esta gramática de tipo 3, ya sabemos que el lenguaje que crea también es de tipo 3. Para ver que llegamos al lenguaje inicial vamos a ir comprobando cada una de las reglas de producción.

Como vemos, nuestro lenguaje tiene el símbolo terminal ‘a’, por lo que debemos incluirlo desde un principio. Por eso la primera regla de producción que creamos es S→aB, con la B lo que buscamos es por obtener $b^i$, para ello creamos la regla de producción B→bB.

Para poder obtener la ‘c’ de nuestro lenguaje creamos dos reglas de producción, B→C y C→cC, estas reglas lo que nos permite es convertir la B en C y así obtener la primera C de nuestro lenguaje y a partir de ahí poder generar la cantidad de C que queramos, consiguiendo así $c^j$.

Por último, para que tengamos el mismo lenguaje debemos poder obtener una ‘d’. Para eso creamos la regla de producción, C→d.

**Así obtenemos el lenguaje: *{$ab^ic^jd$ : 0<=i<=j i,j ∈ N}***

# 2. PRÁCTICA 2.- AND → AD con jFlap

**2.1 Apartado a) Implemente con jFlap un autómata determinista aportándole a la herramienta un autómata no determinista, además hacerlo también con un autómata con transiciones nulas.**

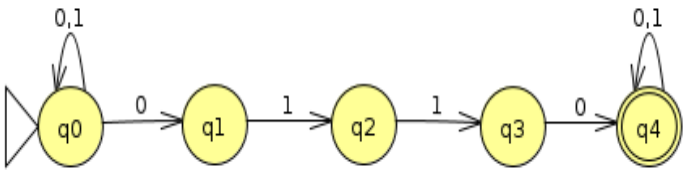


*Figura 1. Autómata Finito No Determinista.*

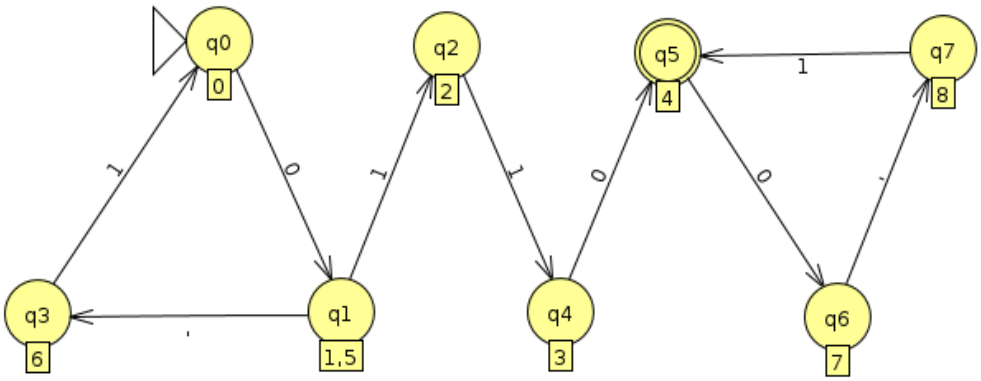


*Figura 2. Resultado de la conversión del autómata de la Figura1 en Autómata Finito Determinista utilizando jFlap.*

### 2.2 Apartado b) Conversión de autómata con transiciones nulas

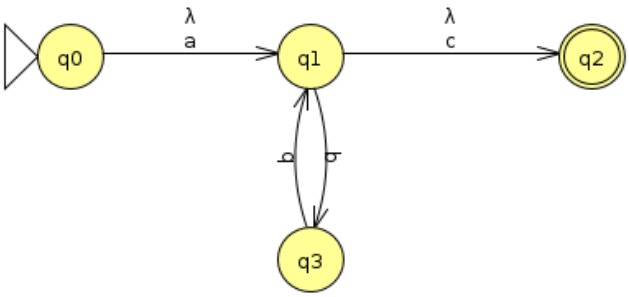


*Figura 3. AFND con transiciones nulas.*

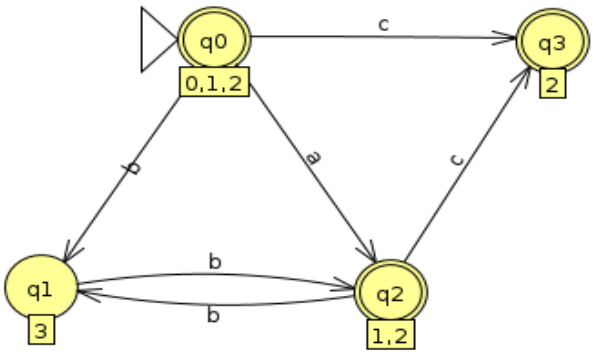


*Figura 4. Resultado de la conversión en AFD.*

# 3. PRÁTICA 3.- Lex (expresiones regulares)

**Crear un fichero lex del código mostrado en el T2. Pág. 216. Compilar dicho fichero lex para generar un archivo .yy.c.**

**Código :**

```
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 de Enteros %d",ent,real,ident,sumaent);return 1;}
```

**Para comprobar su buen funcionamiento vamos a seguir los siguientes pasos:**

1. **Creamos un fichero con el nombre *“codigo”*.**
2. **Ejecutamos en lex el fichero creado en el paso anterior: *“$ lex codigo”***
3. **Compilar el fichero .yy.c creado a partir de lex: *“$ gcc lex.yy.c -o codigo -ll”***
4. **Ejecutar el programa: *“$ ./ejemplo Entrada Salida”***

**Datos fichero Entrada :**

```
3
2
0.33
100
50.65
5.4
a
b
```

**Datos fichero Salida :**

```
Numero entero 3
Numero entero 2
Num. real0.33
Numero entero 100
Num. real50.65
Num. real5.4
Var. ident.a
Var. ident.b
Numero de Enteros3, reales3, ident2,          Suma de Enteros 105
```

# 4. PRÁCTICA 4.- A.P → Gramática

**4.1 Enunciado. Escoger un autómata de pila vacía de los que aparecen en las transparencias del tema 5. Partiendo de uno de ellos obtener la gramática libre del contexto que crea dicho autómata.**

**Obtener la gramática libre del contexto del autómata con pila vacía habiendo elegido aquel autómata que acepta las cadenas con el mismo número de 0 que de 1:**

**M = ({q1}, {0, 1}, {R, X, Y}, δ, q1, R, ∅)**

**δ(q1, 0, R) = {(q1, XR)}**
**δ(q1, 1, R) = {(q1, YR)}**
**δ(q1, 1, X) = {(q1, ε)}**
**δ(q1, ε, R) = {(q1, ε)}**

**δ(q1, 0, X) = {(q1, XX)}**
**δ(q1, 1, Y ) = {(q1, YY)}**
**δ(q1, 0, Y ) = {(q1, ε)}**

**4.2 Solución :**

Lo primero que hago es añadir una nueva regla por cada estado. Esta regla sería S → [q1, Z0, q1]. A continuación añado las reglas para simular aquellas transiciones que no meten nada en la pila. Estas son :

δ(q1, 1, X) = {(q1, ε )} añado [q1, X, q1] → 1

δ(q1, 0, Y) = {(q1, ε )} añado[q1, Y, q1] → 0

δ(q1, ε, R) = {(q1, ε)} añado[q1, R, q1] → ε

Para terminar, añado las reglas que simulan las transiciones del autómata que introducen elementos en la pila. Estas reglas son :

- Para δ(q1, 0, R) = {(q1, XR)} añado [q1, R, q1] → 0[q1, X, q1][q1 , R, q1]
- Para δ(q1, 0, X) = {(q1, XX)} añado [q1, X, q1] → 0[q1, X, q1][q1, X, q1]
- Para δ(q1, 1, R) = {(q1, YR)} añado [q1, R, q1] → 1[q1, Y, q1][q1, R, q1]
- Para δ(q1, 1, Y) = {(q1, YY)} añado [q1, Y, q1] → 1[q1, Y, q1][q1, Y, q1]

La gramática que he obtenido tras estos pasos es :

***[q1, R, q1] → 0[q1, X, q1][q1, R, q1]***
***[q1, R, q1] → 1[q1, Y, q1][q1, R, q1]***
***[q1, X, q1] → 1***
***[q1, Y, q1] → 0***
***[q1, R, q1] → ε***
***[q1, X, q1] → 0[q1, X, q1][q1, X, q1]***
***[q1, Y, q1] → 1[q1, Y, q1][q1, Y, q1]***