Modelos de computación (2015-2016)
Grado en Ingeniería Informática
Universidad de Granada

Memoria Prácticas

- Añadida Práctica 7.
- Añadidos ejemplos con LEX en la Práctica 4.
- He usado JFLAP para la realización y comprobación de algunos ejercicios de esta práctica
- El resto de prácticas fueron revisadas en clase

Francisco Javier Navarro Morales

26 de enero de 2016

Índice

1. Practica 1: G = (V,T,P,S), donde V = S,A,B, T = a,b, el símbolo de partida es S y las reglas son:

$$\mathrm{S} \rightarrow \mathrm{aB}, \quad \mathrm{S} \rightarrow \mathrm{bA}, \quad \mathrm{A} \rightarrow \mathrm{a}, \quad \mathrm{A} \rightarrow \mathrm{aS},$$
$$\mathrm{A} \rightarrow \mathrm{bAA}, \quad \mathrm{B} \rightarrow \mathrm{b}, \quad \mathrm{B} \rightarrow \mathrm{bS}, \quad \mathrm{B} \rightarrow \mathrm{aBB}$$

Esta gramática genera el lenguaje: $L(G) = \{u|u \; \epsilon \; \{a,b\}^{+} \; y \; Na(u) = Nb(u)\}$, es decir, la gramática genera palabras con el mismo número de 'a' que de 'b'.

Podemos extraer las siguientes interpretaciones de las reglas de producción:

- Interpretación de **'A'** $\rightarrow$ Genera palabras con un Símbolo Terminal 'a' de más.
- Interpretación de **'B'** $\rightarrow$ Genera palabras con un Símbolo Terminal 'b' de más.
- Interpretación de **'S'** $\rightarrow$ Genera una cadena con el mismo numero de 'a' que 'b'.

Hay que demostrar dos cosas:

- Todas las palabras generadas por la gramática tienen el mismo número de a que de b.
- Cualquier palabra con el mismo número de a que de b es generada.

Vamos a ir desarrollando cada posibilidad de forma que para cada paso se apliquen todas las reglas de producción posibles, inicialmente tenemos dos posibilidades:

$$^{1}\mathrm{S} \Rightarrow \mathrm{aB}, \qquad ^{2}\mathrm{S} \Rightarrow \mathrm{bA}$$

Vamos a iniciar el desarrollo para la primera:

S $\Rightarrow$ aB $\Rightarrow$ ab, generamos la palabra **ab**.
S $\Rightarrow$ aB $\Rightarrow$ abS $\Rightarrow$ abaB $\Rightarrow$ abab, generamos la palabra **abab**.
S $\Rightarrow$ aB $\Rightarrow$ aaBB $\Rightarrow$ aabB $\Rightarrow$ aabb, generamos la palabra **aabb**.

Continuamos el desarrollo para la segunda posibilidad:

S $\Rightarrow$ bA $\Rightarrow$ ba, generamos la palabra **ba**.
S $\Rightarrow$ bA $\Rightarrow$ baS $\Rightarrow$ babA $\Rightarrow$ baba, generamos la palabra **baba**.
S $\Rightarrow$ bA $\Rightarrow$ bbAA $\Rightarrow$ bbaA $\Rightarrow$ bbaa, generamos la palabra **bbaa**.

Hemos comprobado que podemos conseguir generar cadenas básicas con $N_a(\mathrm{u}) = N_b(u)$, dónde primero hay símbolos terminales 'a' y luego 'b' ó hay símbolos terminales '(ab)$^{+}$', y lo mismo cambiando el orden de 'a' y 'b'. Si queremos generar cualquier cadena perteneciente al lenguaje lo podemos conseguir combinando las anteriores palabras generadas. Por ejemplo generemos una palabra usando todas las reglas de producción:

$$\mathrm{S} \overset{1}{\Rightarrow} \mathrm{a\underline{B}} \overset{8}{\Rightarrow} \mathrm{aa\underline{B}B} \overset{7}{\Rightarrow} \mathrm{aab\underline{S}B} \overset{1}{\Rightarrow} \mathrm{aaba\underline{B}B} \overset{8}{\Rightarrow} \mathrm{aabaa\underline{B}BB} \overset{7}{\Rightarrow} \mathrm{aabaab\underline{S}BB} \overset{2}{\Rightarrow} \mathrm{aabaabb\underline{A}BB}$$
$$\overset{5}{\Rightarrow} \mathrm{aabaabbb\underline{A}ABB} \overset{3}{\Rightarrow} \mathrm{aabaabbba\underline{A}BB} \overset{4}{\Rightarrow} \mathrm{aabaabbbaa\underline{S}BB} \overset{2}{\Rightarrow} \mathrm{aabaabbbaab\underline{A}BB} \overset{3}{\Rightarrow}$$
$$\mathrm{aabaabbbaaba\underline{B}B} \overset{6}{\Rightarrow} \mathrm{aabaabbbaabab\underline{B}} \overset{6}{\Rightarrow} \mathrm{aabaabbbaababb}$$

Podemos apreciar que encima de cada fecha indicamos el número de la regla utilizada (las identificamos contando de izquierda a derecha y de arriba hacia abajo) en este ejemplo hemos usado las 8 reglas de producción que nos brinda la gramática, comprobando que la palabra que genera **'aabaabbbaababb'** contiene el mismo número de símbolos terminales 'a' que 'b', en concreto N = 7. Con este ejemplo demostramos también que combinando las reglas de producción podemos generar cualquier palabra con la peculiaridad de que el número de 'a' coincide con el de 'b'.

2. Practica 2: Determinar si la gramática G = ({S, A, B}, {a, b, c, d}, P, S) genera un lenguaje de tipo 3 donde P es el conjunto de reglas de producción:

$$S \to AB, \qquad A \to Ab, \qquad A \to a, \qquad B \to cB, \qquad B \to d$$

Esta **gramática genera el lenguaje** $L(G) = \{ab^ic^jd : i, j \in \mathbb{N}\}$

Partiendo de los datos del problema vamos a comprobar que esa gramática genera todas las palabras del lenguaje.

$$S \to \underline{A}B \to \underline{A}bB \to \underline{A}bbB \to ab^i\underline{B} \to ab^ic\underline{B} \to ab^icc\underline{B} \to ab^ic^j\underline{B} \to ab^ic^jd$$

Vemos que todas las palabras que genera dicha gramática pertenecen al lenguaje y no genera ninguna que no pertenezca al lenguaje. Podemos sacar la siguiente interpretación:

- A ->genera subcadenas que empiezan por 'a' seguido de un numero i de 'b'.
- B ->genera subcadenas que empiezan por un numero j de 'c' y acaban en 'd'.
- Estamos ante una gramática libre del contexto(de tipo 2) puesto que sus producciones son de la forma "terminal-Variable y Variable-terminal" ó "Variable-Variable". Por lo tanto el lenguaje que genera es también de tipo 2.

Hasta ahora no hemos hecho mas que comprobar que los datos del problema son correctos, el siguiente paso es ver si conseguimos modificar la gramática, en concreto sus producciones, para conseguir que esta genere un lenguaje regular. Podemos intentar esto porque no hay relación numérica entre los símbolos terminales del lenguaje.

Según la **Jerarquía de Chomsky** un lenguaje es regular si las reglas de producción son de tipo 3, para esto las producciones de la gramática deben ser del tipo: $A \to uB \,||\, A \to u$.

Vamos a usar las siguientes reglas de producción:

S → aB, B → bB, B → C, C → cC, C → d

- S → aB: S es el símbolo inicial. Esta prod. genera la primera "a" y da paso a la variable "B".
- B → bB: Generar un numero i de "b".
- B → C: Pasa a la variable "C".

- C → cC: Genera un número j de ”c”.
- C → d: Para finalizar esta producción nos permite colocar el último símbolo terminal ”d” necesario para formar palabras correctas.

En conclusión hemos encontrado unas reglas de producción de tipo 3, que hacen la gramática regular, capaces de generar el mismo lenguaje que generaban las reglas de producción de partida. **Demostramos que la nueva gramática es de tipo 3 por lo tanto el lenguaje también lo es.**

3. Practica 3: Diseñar un autómata finito determinístico que acepte cadenas que contienen las subcadenas '0110' y '1000'. Las subcadenas pueden aparecer juntas o separadas, también pueden contener más símbolos (0+1) delante o detras de ellas.

Para comenzar esta práctica lo más sencillo es **diseñar el autómata no determinístico** que acepta las cadenas que nos pide el ejercicio. Nos ayudamos de los ejemplos de los autómatas que son capaces de reconocer palaras que contienen las subcadenas '0110' ó '1000, de esta forma conseguimos diseñar el autómata que necesitamos de forma muy fácil, para ello **seguimos los siguientes pasos**:

- Q4 deja de ser un nodo final y P0 deja de ser un nodo inicial.
- Unimos los dos autómatas añadiendo una transición nula entre Q4 y P0.

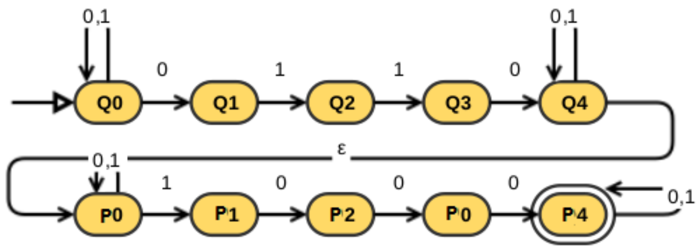


Figura 3.1: Autómata Finito No Determinístico(AFND)

Los autómatas no determinísticos son muy ineficientes porque exploran todas las posibilidades. Para resolver el problema que tenemos entre las manos de una forma más eficiente vamos a transformarlo en un autómata que para cada símbolo leído sepa que acción debe realizar. Es un proceso exponencial a medida en que crece el número de nodos en el 'AFND', para no equivocarnos vamos a realizar una tabla donde indiquemos para cada nodo cual es la acción que realiza al leer cierto símbolo de la cinta de entrada.

Símbolo	**Q0**	**Q1**	**Q2**	**Q3**	**Q4**
0	{Q0,Q1}	{Ø}	{Ø}	{Q4,P0}	{Q4,P0}
1	{Q0}	{Q2}	{Q3}	{Ø}	{Q4,P0,P1}

Tabla 3.1: Tabla nodos Q.

Símbolo	**P0**	**P1**	**P2**	**P3**	**P4**
0	{P0}	{P2}	{P3}	{P4}	{P4}
1	{P0,P1}	{Ø}	{Ø}	{Ø}	{P4}

Tabla 3.2: Tabla nodos P.

Para construir nuestro autómata determinístico nos apoyaremos de las tablas que hemos creado. **Comprobamos para cada estado** de los que tenemos entre **{<estados>}** a que **estado(s) pasa** según lea un 0/1, si tenemos un **estado compuesto** por 2 o más estados tenemos que realizar el proceso para cada subestado y el **resultado será la unión** de los resultados de cada subestado. Una vez generado el nuevo estado tenemos **dos posibilidades** que el **nuevo estado se repita**, en este caso trazamos una flecha desde el estado actual hasta el estado obtenido(etiquetando dicha flecha con el símbolo que produce el cambio de estados); por otra parte, si **es un nuevo estado** debemos colocarlo unidos por una flecha etiquetada. **Repetimos** todo el proceso hasta que todos los estados tengan un estado objetivo para cada símbolo posible(en nuestro caso 0/1).

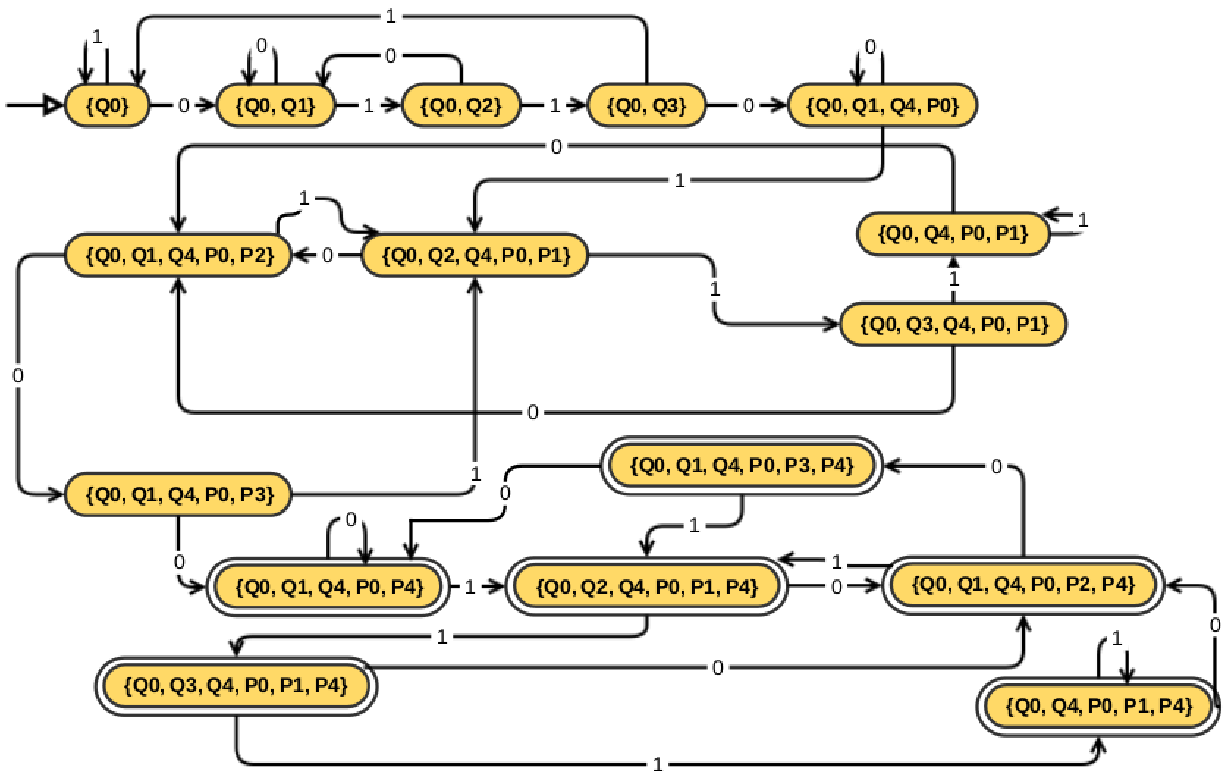


Figura 3.2: Autómata Finito Determinístico(AFD)

4. Practica 4: Realizar programas usando la herramienta de reconocimiento de expresiones regulares LEX

Este programa de ejemplo es capaz de reconocer números(tanto reales como enteros) e identificadores formados por al menos un carácter seguido de más caracteres o dígitos. En el caso de los enteros también lleva la suma de estos que al final muestra junto al numero de reales, enteros e identificadores reconocidos.

Escribe el siguiente código en un fichero llamado `ejemplo.c`:

```
/*Definir Variables Globales*/
%{
int ent=0, real=0, ident=0, sumaent=0, i=0;
%}
/*Definir Expresiones Regulares*/
car      [a-zA-Z]
digito   [0-9]
signo    (\-|\+)
suc      ({digito}+)
enter    ({signo}?{suc})
real1    ({enter}\.{digito}*)
real2    ({signo}?\.{suc})
/*Definir Acciones*/
%%
{enter}                 {ent++; sscanf(yytext,"%d",&i); sumaent +=
    i; printf("Numero entero: %s\n\n",yytext);}

({real1}|{real2})       {real++; printf("Num. real: %s\n\n",yytext
   );}

{car}({car}|{digito})*  {ident++; printf("Var. ident: %s\n\n",
   yytext);}

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

Ahora compila usando `lex` y `gcc`:

```
$ lex ejemplo
$ gcc lex.yy.c -o prog -ll
```

Por último ejecutamos de la siguiente forma:

```
$ ./prog < 'fichero_entrada' > 'fichero_salida'
```

Implementación del autómata de la *"Practica 3: Diseñar un autómata finito determinístico que acepte cadenas que contienen las subcadenas '0110' y '1000'"* :

```
/*Definicion de variables, includes, etc*/
%{
int i=0,t=0;
char id[]=" ";
%}

/*Definicion de expresiones regulares*/
car      [a-zA-Z]
digito   [0-9]
patron   (0|1)*0110(0|1)*1000(0|1)*

/*Cuando encuentre una expresion regular hace la funcion
   correspondiente*/
%%

{car}({car}|{digito})*  {t++; sscanf(yytext,"%s",id);}
        {patron}        {i++; printf("\nCadena con id %s -> %s es
            una cadena valida.",id, yytext);}
.|\n {;}

%%

/*Cuando acabe de leer el fichero se ejecuta la seccion yylex (
   main) ejecuta la funcion yywrap que muestra la informacion*/
yywrap(){
        i?printf("***Hay %d/%d aceptadas***\n",i,t):printf("\nNo
            hay cadena(s) valida(s)\n");
        return 1;
}
```

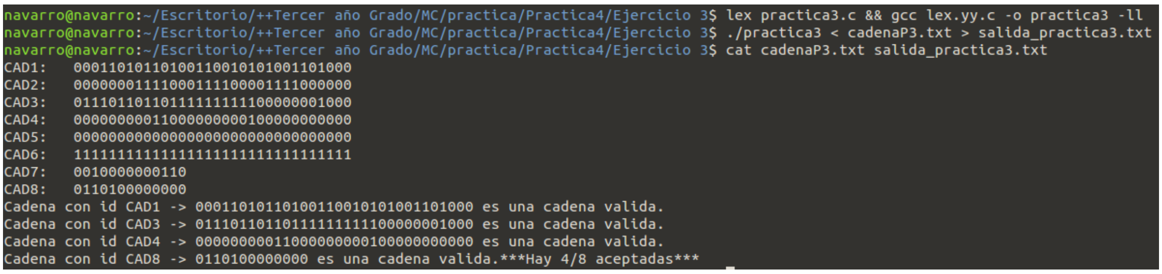


Figura 4.1: Captura de ejecución del programa

Implementación de un reconocedor para saber si un email tiene un formato correcto (nombre @ dominio) :

```
/*Definicion de variables, includes, etc*/
%{
int ent=0;
%}

/*Definicion de expresiones regulares*/
car      [a-zA-Z]
digito   [0-9]
signo    (\-)
iden     ({car}|{digito})(([_\.\-]?({car}|{digito})+)*)
dominio @([A-Za-z0-9]+)(([\.\-]?[a-zA-Z0-9]+)*)\.([A-Za-z]{2,})
email    ^{iden}{dominio}$

/*Cuando encuentre una expresion regular hace la funcion correspondiente*/
%%

{email}          {ent++; printf("\nEl email %s tiene un formato valido(texto@dominio) \n", yytext);}

.|\n {;}

%%

/*Cuando acabe de leer el fichero se ejecuta la seccion yylex (main) ejecuta la funcion yywrap que muestra la informacion*/
yywrap(){
        printf("\nNumero de email validos: %d\n %s",ent,yytext);
                return 1;
}
```

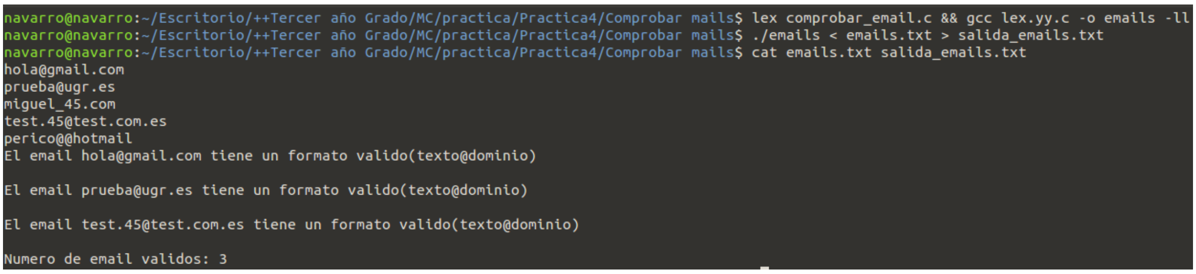


Figura 4.2: Captura de ejecución del programa

Implementación de sumador de dos números romanos :

```
WS  [ \t]+

%%
int total=0;

I   total += 1;
IV  total += 4;
V   total += 5;
IX  total += 9;
X   total += 10;
XL  total += 40;
L   total += 50;
XC  total += 90;
C   total += 100;
CD  total += 400;
D   total += 500;
CM  total += 900;
M   total += 1000;

{WS}     |
\n  return total;
%%
int main (void) {
int first, second;

first = yylex ();
second = yylex ();

printf ("%d + %d = %d\n", first, second, first+second);
return 0;
}
```

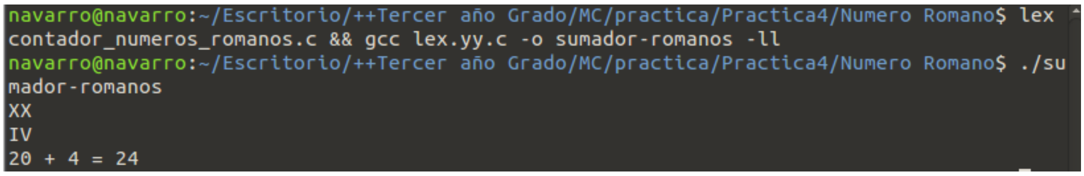


Figura 4.3: Captura de ejecución del programa

5. Practica 5: Dado L = { 0 U 1 | U $\in$ {0,1}* } obtener:

1. **Expresión Regular**
2. **Autómata Finito Determinístico**
3. **Gramática lineal por la derecha y por la izquierda**

1) La expresión regular correspondiente al lenguaje dado es '**0(0+1)*1**'.

2) Para obtener el Autómata Finito Determinístico(AFD) nos ayudamos de la expresión regular que tenemos formada, de ella podemos extraer que necesitamos reconocer cadenas que comiencen por un 0 seguida de n veces 0 ò 1 (con n>=0) y acaben en 1.

Vamos a dar un paso antes de obtener el AFD, se trata de obtener un autómata que nos resuelva el problema aunque este no sea determinístico.

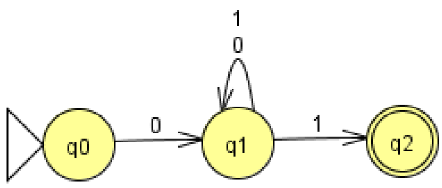


Figura 5.1: Autómata Finito NO Determinístico(AFND)

Y ahora como aprendimos en la práctica3 pasaremos este 'AFND' a un 'AFD'.

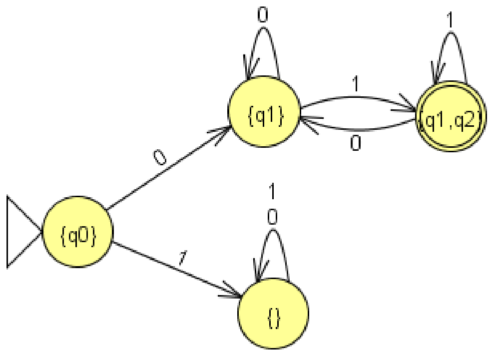


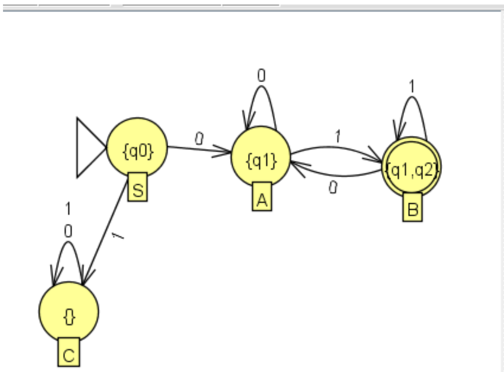
Figura 5.2: Autómata Finito Determinístico(AFD) *{}:Estado Error*

3) Siguiendo los pasos dados por el problema podemos generar una gramática lineal por la derecha a partir del autómata determinístico:

- Cada estado determina una variable de la gramática.

- Para cada transición entre estados añadimos una producción. En la parte izquierda ponemos el estado añadiendo en la parte derecha el símbolo leído + el estado al que se dirige.
- Añadir una producción V ->λ, para cada V siendo esta estado final.

La gramática asociada a este autómata es(q0-inicial):

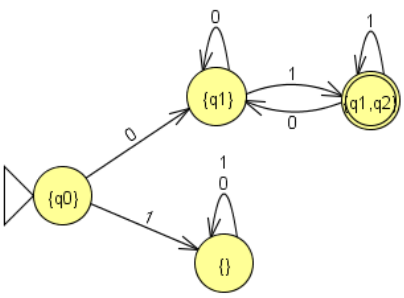


LHS		RHS
B	$\rightarrow$	1B
C	$\rightarrow$	1C
C	$\rightarrow$	0C
A	$\rightarrow$	0A
S	$\rightarrow$	1C
B	$\rightarrow$	λ
S	$\rightarrow$	0A
A	$\rightarrow$	1B
B	$\rightarrow$	0A

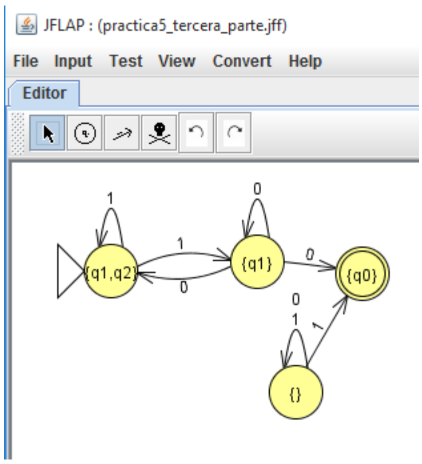
Figura 5.3: AFD - Gramática Lineal por la derecha

Etiquetamos los nodos con una letra para generar una gramática de forma más clara

Llegados hasta este paso solo nos falta obtener la gramática lineal por la izquierda, para ello invertimos el autómata(cambiando la dirección de las transiciones entre estados y cambiamos los estados finales por los iniciales), construimos la gramática lineal por la derecha asociada e invertimos la parte derecha de las producciones.



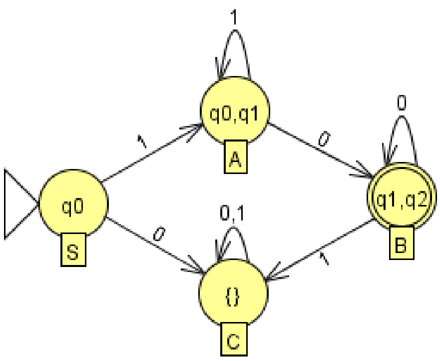
(a) AFD de partida



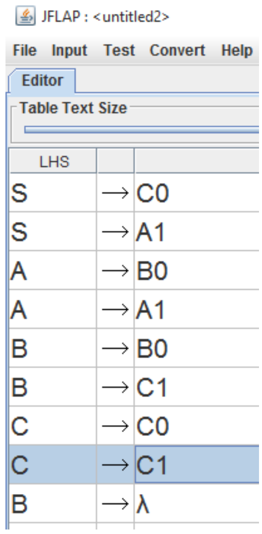
(b) AF invertido

Figura 5.4: Pasando GLD a GLI

Al invertir el diagrama de la figura (a) obtenemos el de la figura (b) resultante que es un diagrama no determinístico, por lo cual necesitamos pasarlo a determinístico. Renombramos los estados, pasamos a determinístico y obtenemos la gramática lineal por la derecha asociada al AFD, por último giramos la parte derecha de las producciones, resultado:



(a) AFD de partida



(b) AFD de partida

Figura 5.5: AF – Gramática Lineal Izquierda

6. Practica 6: Elegir una gramática libre del contexto que contenga producciones nulas e unitarias. Aplicar trasformaciones para dejarlas en Forma Normal de Chomsky. Opcionalmente pasar a Forma Normal de Greibach.

Sea la gramática:

$$S \to SS, \quad S \to CA, \quad A \to bAA, \quad A \to aC, \quad A \to B,$$
$$B \to aSS, \quad B \to BC, \quad C \to CC, \quad C \to \varepsilon$$

El primer paso para normalizar la gramática es eliminar las producciones nulas siguiendo los siguientes pasos:

1. Marcar (para eliminar) todas las producciones nulas del tipo A→ ε.
2. Para cada producción que en su parte derecha tuviese como mínimo una de las producciones que hemos marcado, tenemos que añadir todas las producciones que podríamos conseguir usando la producción eliminada.
3. Eliminar todas las producciones que marcamos en el primer paso.

Aplicando el primer paso del algoritmo..

$$S \to SS, \quad S \to CA, \quad A \to bAA, \quad A \to aC, \quad A \to B,$$
$$B \to aSS, \quad B \to BC, \quad C \to CC, \quad C \to \varepsilon$$

Aplicando el segundo paso del algoritmo..

$$S \to SS, \quad S \to CA, \quad A \to bAA, \quad A \to aC, \quad A \to B,$$
$$B \to aSS, \quad B \to BC, \quad C \to CC, \quad C \to \varepsilon,$$
$$S \to A, \quad A \to a, \quad B \to B, \quad C \to C$$

Por último aplicando el tercer paso del algoritmo obtenemos una gramática equivalente sin producciones nulas:

Gramática sin producciones nulas

$$S \to SS, \quad S \to CA, \quad A \to bAA, \quad A \to aC,$$
$$A \to B, \quad B \to aSS, \quad B \to BC, \quad C \to CC,$$
$$S \to A, \quad A \to a, \quad B \to B, \quad C \to C$$

Seguimos procesando la misma gramática para llegar a nuestro objetivo final. Ahora queremos eliminar las transiciones unitarias, siguiendo los siguientes pasos:

1. Nos ayudamos de un conjunto H donde introduciremos para de valores siguiendo los siguientes criterios:
   a) Para cada producción del tipo A→B, añadimos en H el par (A,B).
   b) Si en H encontramos pares del tipo (S,A)-(A,B), añadimos el par (S,B) en H.
2. Eliminamos las producciones unitarias.
3. Para cada par en H realizamos el siguiente proceso, suponiendo que tenemos el par (A,B) tenemos que añadir una producción por cada producción del tipo tipo B→, tal que A→<parte derecha de las producciones que sean del tipo B→. Consiguiendo generar por A las producciones que se generaban desde B.

Aplicando el algoritmo..

S→SS, S→CA, A→bAA,
A→aC, A→B, B→aSS,
B→BC, C→CC, S→A,
A→a, B→B, C→C

H=(A,B),(S,A),(B,B),(C,C),(S,B)

Quitar A→B y añadir:

- A→aSS, A→BC

Quitar S→A y añadir:

- S→bAA, S→aC, S→a

Quitar S→B y añadir:

- S→aSS, S→BC

Para B→B y C→C no hay producciones que añadir.

Gramática sin producciones nulas, ni unitarias

S→SS, S→CA, S→bAA, S→aC, S→a,
S→aSS, S→BC, A→bAA, A→aC, A→a,
A→aSS, A→BC, B→aSS, B→BC, C→CC

Pasar a Chomsky:
Sea la gramática:

$$\underline{S \to SS}, \quad S \to bAA, \quad \underline{S \to a}, \quad S \to aSS,$$
$$A \to bAA, \quad \underline{A \to a}, \quad A \to aSS,$$

Las que aparecen subrayadas están ya en Forma Normal de Chomsky

El resto de producciones las tenemos que transformar hasta que sean de la forma:

- A→aα, donde a es un símbolo terminal y α pertenece a V*
- A→ α, donde α pertenece a V* siempre que $\alpha \geq 2$

Sustituimos S→bAA por:

- S→C_bAA

Sustituimos A→bAA por:

- A→C_bAA

Añadir:

- $\underline{C_b \to b}$

Sustituimos S→aSS por:

- S→C_aSS

Sustituimos A→aSS por:

- A→C_aSS

Añadir:

- $\underline{C_a \to a}$

Continuamos con el refinamiento:

Sustituimos S→C_bAA por:

- S→$C_b D_1$

Sustituimos A→C_bAA por:

- A→$C_b D_1$

Añadir:

- D_1→AA

Sustituimos S→C_aSS por:

- S→$C_a D_2$

Sustituimos A→C_aSS por:

- A→$C_a D_2$

Añadir:

- D_2→SS

Gramática en Forma Normal de Chomsky

S→SS, S→a, A→a, S→$C_b D_1$, S→$C_a D_2$, A→$C_b D_1$, A→$C_a D_2$, C_a →a, C_b →b, D_1→AA, D_2→SS,

7. Practica 7: Obtener la gramática a partir de un autómata con pila

$$\textbf{Por pila vacía. } M = (q1, q2, 0, 1, R, X, \delta, q1, R, \theta)$$
$$\delta(q1, 0, R) = (q1, XR) \qquad \delta(q1, 0, X) = (q1, XX)$$
$$\delta(q1, \epsilon, R) = (q1, \epsilon) \qquad \delta(q1, 1, X) = (q2, \epsilon)$$
$$\delta(q2, 1, X) = (q2, \epsilon) \qquad \delta(q2, \epsilon, R) = (q2, \epsilon)$$

La interpretación de los estados es la siguiente:

- En el estado q1 leemos 0 de la cinta de entrada escribiendo una X en la pila.
- En el estado q2 leemos 1 haciendo matching con las X de la pila.

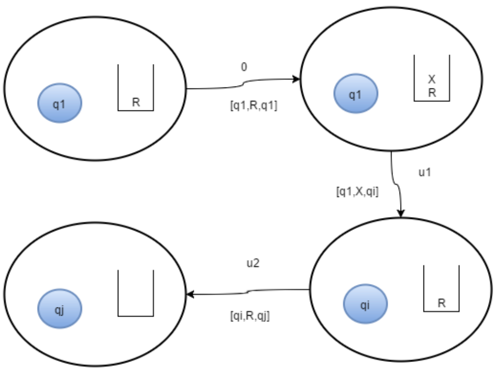
Ya estamos listos para comenzar a extraer las producciones de la gramática.

En primer paso añadimos una transición del tipo S → [q0,z0,q] siendo q0 el estado inicial, z0 el símbolo inicial de la pila y qϵQ.

S → [q1,R,q1]   S → [q1,R,q2]

En segundo paso para cada transición que lea y escriba en la pila tenemos que añadir una producción por cada estado hasta dejar la pila vacía. Este paso queda más claro con el siguiente ejemplo:

δ(q1,0,R) = (q1,XR)



[q1,R,q1]→0[q1,X,q1][q1,R,q1]
[q1,R,q1]→0[q1,X,q2][q2,R,q1]
[q1,R,q2]→0[q1,X,q1][q1,R,q2]
[q1,R,q2]→0[q1,X,q2][q1,R,q2]

δ(q1,0,X) = (q1,XX)

[q1,X,q1]→0[q1,X,q1][q1,X,q1]
[q1,R,q1]→0[q1,X,q2][q2,X,q1]
[q1,R,q2]→0[q1,X,q1][q1,X,q2]
[q1,R,q2]→0[q1,X,q2][q1,X,q2]

δ(q1, ϵ,R) = (q1, ϵ)

[q1,R,q1]→ϵ

δ(q2, ϵ,R) = (q2, ϵ)

[q2,R,q2]→ϵ

δ(q2,1,X) = (q2, ϵ)

[q2,X,q2]$\rightarrow$1

δ(q1,1,X) = (q2, ϵ)

[q1,X,q2]$\rightarrow$1

Y de esta forma obtenemos la gramática asociada al lenguaje L = $0^i 1^i : i \geq$ 0