Grupo Miércoles 10:00-12:00 semanas B

—Práctica 2—

Autor: Athanasios Usero

NIP: 839543

NOTA:

Se va a emplear, a la hora de trabajar con expresiones regulares, la notación que se utiliza en *Flex,* con el fin de hacer más fácil la posterior lectura de los ficheros .l .

Ejercicio 1:

1. Resumen

El siguiente problema presenta la singularidad de plantearse sobre un fichero de flex correctamente diseñado, el cual tiene tres secciones bien distinguidas. Esto induce al planteamiento de una resolución basada en condiciones de arranque.

En primer lugar, se han definido cinco condiciones de arranque exclusivas (puesto que cada sección trabajará de una manera diferente): reglas (parte de la sección de reglas que no es acción), que servirá para contabilizar el número total de reglas; usuario, para contabilizar el total de líneas de la sección código usuario; acciones, que servirá para contabilizar instrucciones y facilitar el reconocimiento de reglas; comentario, para contabilizar comentarios y tratarlos de manera diferente al resto de secciones; y código, para representar bloques de código dentro de las acciones. Además, la condición de arranque inicial servirá para analizar la sección de declaraciones.

Cada vez que se reconozca un fin de línea, incrementará el contador de línea, dependiendo de la condición de arranque. Así, para con la condición de arranque INITIAL, aumentarán las líneas de la sección de declaraciones; cuando estén activas reglas, acciones o código, aumentará el número de líneas de la sección de reglas; mientras que se active finalmente la condición de arranque usuario, contabilizará el número de líneas de la sección código usuario. Por otro lado, cuando una línea comience con “%%”, se procederá a activar la condición de la siguiente sección (reglas o usuario), y se restará 1 al número de líneas de dicha sección. Esto se debe a que esa línea contabilizará para la nueva sección, pero las líneas delimitadoras no cuentan en ningún caso.

La condición de arranque reglas activa los siguientes patrones a reconocer como reglas:

$$r_1 = \text{^[^]} \quad (1)$$
$$r_2 = \text{"\{"} \quad (2)$$

Mientras que (1) sirve para contabilizar reglas, pues reconoce todo carácter diferente al espacio en blanco al inicio de una línea (fuera de la sección de acciones), siendo así como empieza toda regla. Además cuando se reconozca una llave izquierda (2), se activará la condición de arranque acciones.

La sección de acciones queda aislada por dos condiciones de arranque, acciones y código. La condición de código se ha propuesto para evitar que cuando un bloque de código se cierre con una llave esta no se interprete como el fin de una acción. Así, cuando se reconozca (2), se activará la condición de código y una variable contadora de llaves izquierdas incrementará a 1. Con esta condición activa, cada vez que se reconozca "{"se incrementará el contador de llaves izquierdas, mientras que si se reconoce "}" incrementará el contador de llaves derechas. Estos dos contadores se deben a que en un mismo bloque de código pueden aparecer otros bloques anidados. Por ellos, al último patrón se le añade la instrucción, en caso de que el número de llaves izquierdas sea igual al de derechas, de actualizarlas a 0 y reactivar la condición de acciones, puesto que en este caso cualquier bloque de código se habrá cerrado. Del mismo modo, cuando en la sección de acciones se reconozca "}", se reactivará la condición de reglas.

Para contabilizar instrucciones, se tiene en cuenta que cualquier ";" en una cadena no cuenta como tal, por lo que se ha propuesto el siguiente patrón:

$$r_3 = [\text{"}].*[\text{"}] \qquad (3)$$

El cual consume toda cadena sin permitir consumir sus símbolos en otra regla. Esto permite que al reconocer ";" en una sección de acciones o código se incremente de forma segura el contador de instrucciones. Además, dado que un comentario tiene una forma definida, y siempre termina con un final de línea, se han propuesto los siguientes patrones para reconocerlos y tratarlos:

$$r_4 = \text{"//"} \qquad (4)$$
$$r_5 = .* \$ \qquad (5)$$

La regla (4) estará activa con cualquier condición de arranque (excepto el propio comentario), y a esta le asociamos la acción de activar la condición comentario e incrementar el contador de este, pues siempre que aparezcan dos barras vendrán asociadas con una sección de comentarios. Con (5), consumimos toda cadena de un comentario (lo que sirve para no contabilizar ";" como instrucción, y reactivamos la condición de arranque anterior, que se ha almacenado en una variable. Cabe notar que no se consume el símbolo de final de línea, pues este será posteriormente reconocido por la sección del fichero de flex correspondiente.

2. Pruebas

El fichero de entrada a analizar constará de un programa sintácticamente correcto de Flex. En este se han escrito 8 líneas para la sección de declaraciones, 10 en la sección de reglas y 3 en la sección de código usuario. Además, para comprobar el correcto funcionamiento de las reglas para analizar comentarios, se ha dispuesto al menos uno por sección (declaraciones, línea 1 y 4; transición, línea 9; reglas, línea 19; acciones, línea 13; código, línea 16; y usuario, línea 21). Además, en la sección de reglas se han creado 5 reglas, que en total conllevan 8 instrucciones, bien dentro de bloques de código o no. Además, en las líneas 10 y 18 se han puesto dos tipos de cadenas que pueden aparecer con un punto y coma como parte de ellas, comprobando así que no quedan reconocidas como instrucciones.

ENTRADA:

```
 // Athanasios Usero, NIP¶
%{¶
#include < math.h>¶
//comentario;;;; ¶
int a = 0; ¶
char c; ¶
%}¶
¶
%% //otro comentario¶
t+ {printf ("hola ;"); a++;}¶
ggg  {}¶
"aabb"[A-Z]+ { a++; if (a ==0) {a ++;}¶
      // otro comentario;;; ¶
return a;} ¶
jjj {if(a==0){a++; if(true){return a;} ¶
   // Comentario dentro de bloque de código ¶
}}¶
f  {return (";" == c);} ¶
 //comentario en sección de reglas¶
%%¶
int main () { ¶
//y otro comentario¶
  yylex ();¶
}¶
```

SALIDA:

L1:8¶

L2:10¶

L3:3¶

C:7¶

R:5¶

I:8¶

Ejercicio 2:

1. Resumen

Para implementar el siguiente problema, primero debemos razonar el algoritmo que nos permita obtener todas las secuencias de reinicio completo, y después crear reglas en Flex con la información obtenida.

Dado un autómata finito determinista, queremos obtener:

$$RC(M) = \{w \in \Sigma^* \mid w\ es\ una\ secuencia\ de\ reinicio\ completo\ de\ M\}$$

donde $w \in \Sigma^*\ SRC \rightarrow \forall\ q \in Q \quad \delta^*(q, w)$.

Debemos considerar, en este sentido, todas las cadenas partiendo de cada estado, considerando así todos los estados como estados iniciales. Por ello, se añade un nuevo estado inicial q0' a M, que se conectará mediante ϵ transiciones a todos los estados de M. Así, sin necesidad de consumir ningún símbolo, nos podremos situar en cualquier estado de M.

Partiendo de este nuevo AFnD M', tenemos que comprobar si, en efecto, existe alguna cadena de reinicio completo. Sabemos que al determinizar un AFnD con el algoritmo del conjunto potencia de los estados, los nuevos estados serán subconjuntos del subconjunto de estados Q de M'. Estos subconjuntos representan variantes de estados que comparten una misma sucesión de símbolos. Por ellos, si uno de estos subconjuntos esta conformado únicamente por el estado inicial de M q0, significará que existen cadenas que parten desde cualquier estado de M y finalizan exclusivamente en el estado inicial, quedando así comprobada la existencia de secuencias de reinicio completo.

Ahora, con el nuevo AFD Md' obtenido, si definimos su conjunto Fd' como el estado q0, habremos construido un autómata que tan solo reconozca estas cadenas que lleguen al estado inicial desde cualquier otro estado. Podemos también minimizar el autómata finito determinista, con el fin de simplificar la expresión regular que lo describe.

Con todo esto, mostramos el proceso de obtención del secuencias de reinicio completo para los autómatas M1 y M2:

- Autómata 1:

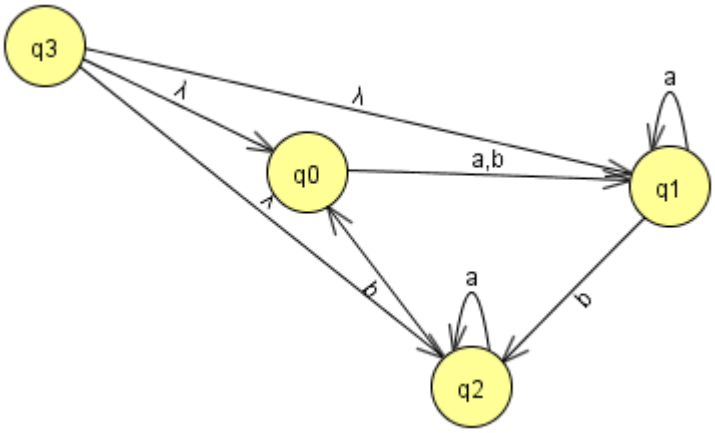

Ilustración 1. Paso 1: AfnD con inicio en todo estado

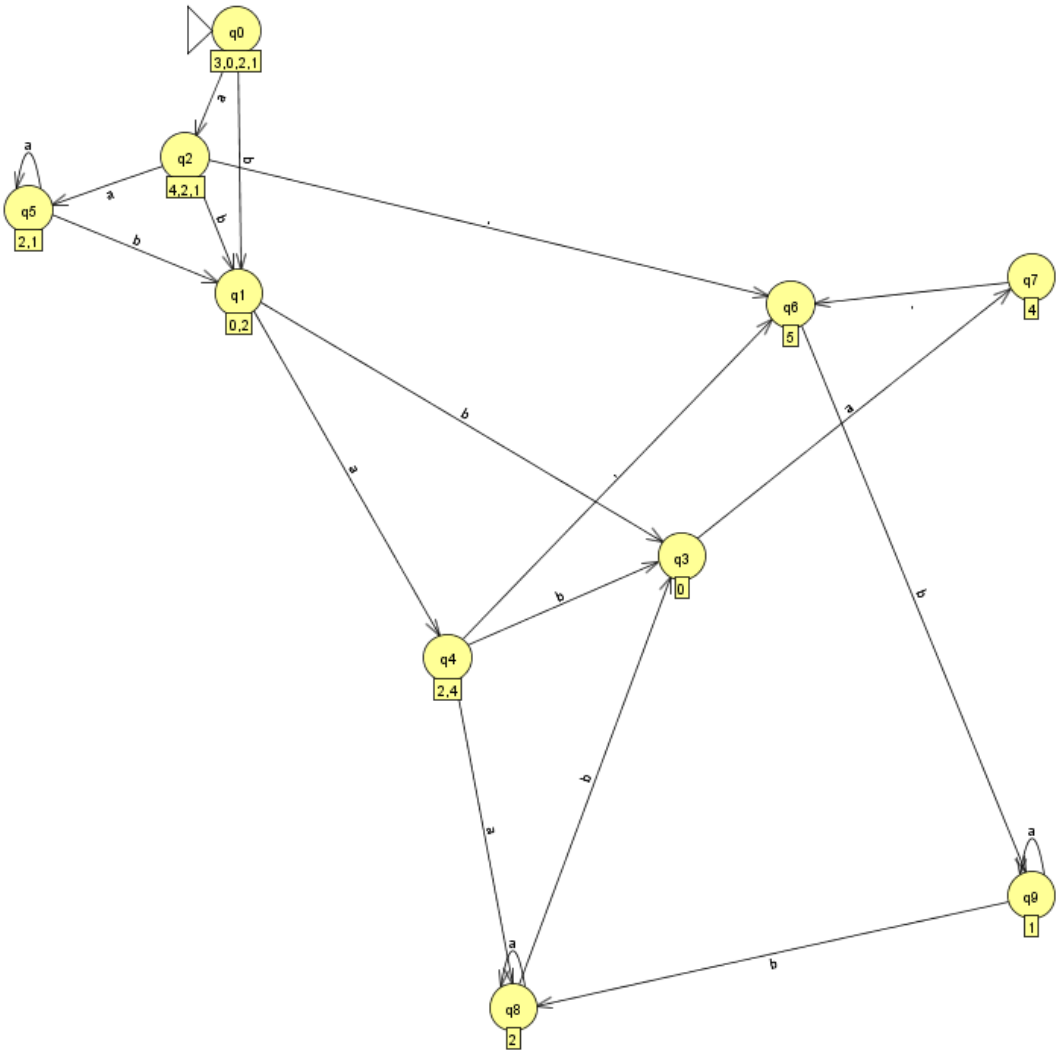

Ilustración 2. Paso 2: Determinización de un AFnD

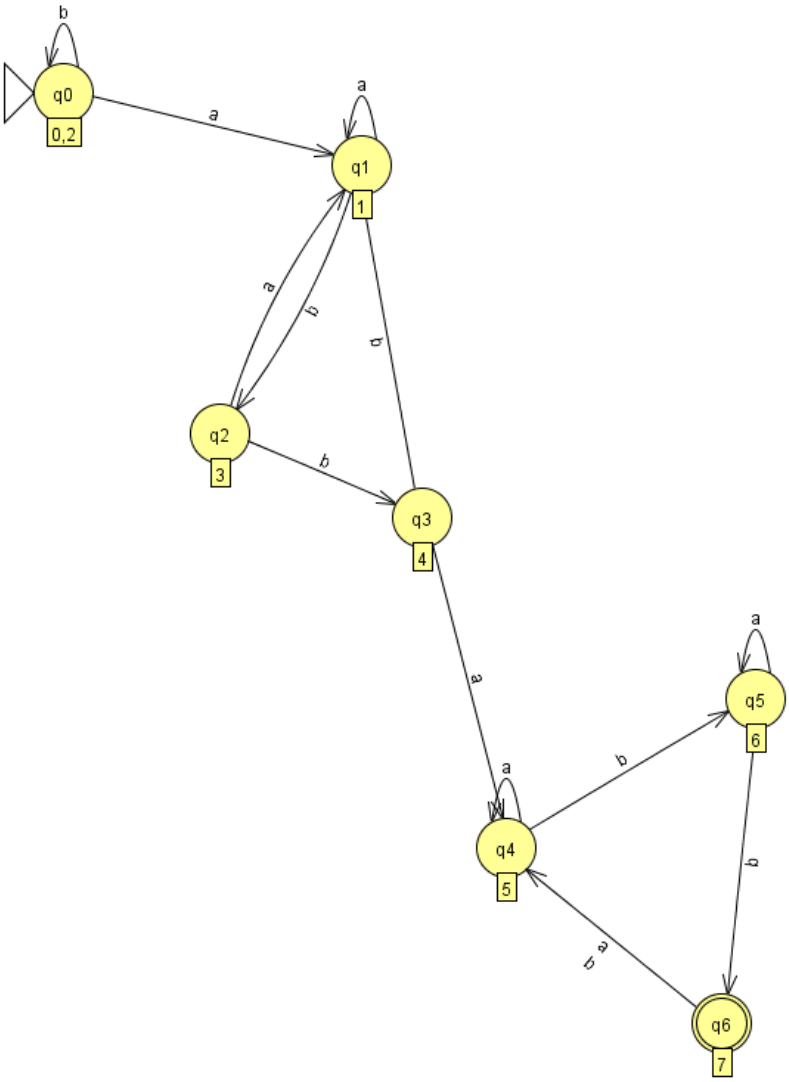

Ilustración 3. Paso 3: Minimización del AFD

- Autómata 2:

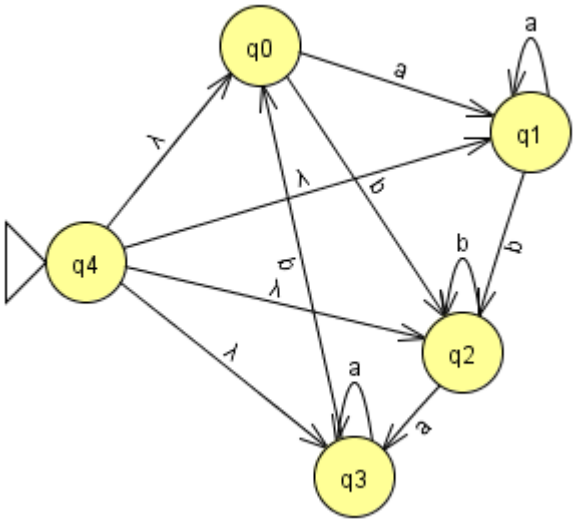

Ilustración 4. Paso 1: AFnD con inicio en todo estado

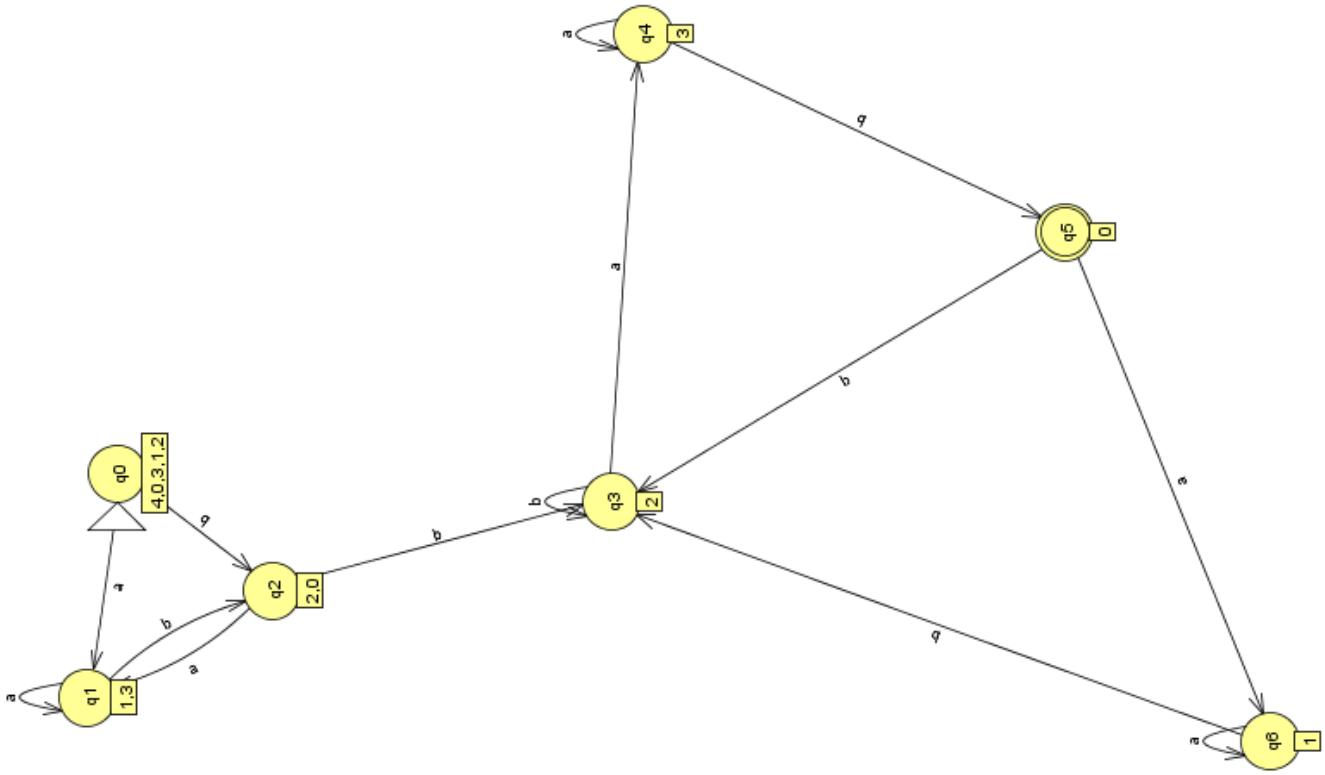

Ilustración 6 Paso 2: Determinización de un AFnD

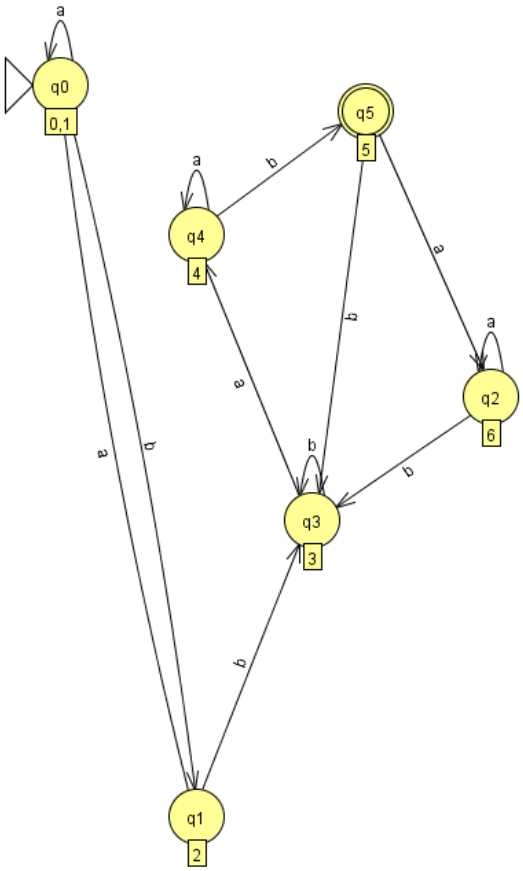

Ilustración 5 Paso 3: Minimización del AFnD

Con todo esto dispuesto, el programa en Flex solo constará de dos reglas, las dos expresiones regulares que generan el lenguaje de M1 y M2. Estas son, respectivamente:

$$r_1 = [b]^*a[a]^*b(a[a]^*b)^*b(b[a]^*b(a[a]^*b)^*b)^*a[a]^*b[a]^*b\big((a|b)[a]^*b[a]^*b\big)^* \quad (6)$$

$$r_2 = \big((a)\big|(ba)\big)^*bb[b]^*a[a]^*b\left(\big(b\big|(a[a]^*b)\big)[b]^*a[a]^*b\right)^* \quad (7)$$

Con estas dos reglas, y la acción correspondiente a escribirlas por pantalla con asteriscos o guiones en cada caso, se podrían reconocer secuencias de reinicio completo de cada autómata, pero sin contemplar la existencia de secuencias de ambos autómatas. Con este fin, al patrón (6) se le asocia la instrucción especial REJECT, que volverá a disponer la cadena para que la segunda regla de mayor prioridad la procese, y al patrón (7) sí que se le asocia la instrucción de escribir la cadena por pantalla. De este modo, cuando una cadena pertenezca a M1 y M2, primero se la reconocerá por la regla (6), y después se volverá a disponer de ella para que se formatee acorde al autómata M2. En caso de que se solamente pertenezca a M1, se volverá a disponer de esa cadena y quedará reflejada en el fichero de salida.

2. Pruebas

Para comprobar su correcto funcionamiento, se ha dispuesto de un fichero de texto en el que en las líneas 2, 7 y 8 aparecen las secuencias de reinicio completo más cortas de los autómatas M1, M2, y su intersección. En las líneas 9,10 y 11 se han escrito cadenas arbitrariamente largas que pertenezcan a uno de los autómatas; mientras que en el resto de líneas se han escrito cadenas con formato incorrecto o que no constituyan secuencias de reinicio completo.

ENTRADA:

```
abbabbh¶
abbabb¶
abbabbh¶
abbabb ab¶
ab babb¶
abbabb ¶
bbab¶
abbabbbbab¶
bbbaaaabaaaabaaaaaabbbaaaaaaabababbaaaabbaaaaabaabbabb¶
aaaaaabababaaabbbbaabaaaabbbaabbabaaab¶
aaaaaabababaaabbbbaabaaaabbbaabbbaaab¶
abababbababababbababbbbabbababababababbbbbababababbababababbbbababab¶
```

SALIDA:

```
abbabbh¶
**abbabb¶
abbabbh¶
abbabb ab¶
ab babb¶
abbabb ¶
--bbab¶
**--abbabbbbab¶
**bbbaaaabaaaabaaaaaabbbaaaaaaabababbaaaabbaaaaabaabbabb¶
**aaaaaabababaaabbbbaabaaaabbbaabbabaaab¶
--aaaaaabababaaabbbbaabaaaabbbaabbbaaab¶
abababbababababbababbbbabbababababababbbbbababababbababababbbbababab¶
```