

PRÁCTICAS MODELOS DE COMPUTACIÓN

Índice

1. Práctica 1.....	2
1. Presentación del problema.....	2
2. Primera aproximación.....	2
3. Obtención del autómata a partir de una expresión regular.....	2
3.1. Problema del autómata inicial.....	2
3.2. Obtención de la expresión regular.....	2
3.2. Obtención del autómata finito no determinístico a partir de la expresión regular.....	4
4. Test.....	5
4.1. Ejecución múltiple.....	5
4.2. Ejecución paso a paso.....	6
2. Práctica 2.....	7
1. Presentación del problema.....	7
2. Solución.....	7
2.1. Elección del lenguaje regular.....	7
2.2. Obtención del autómata.....	7
2.3. Representación del autómata final.....	8
2.4. Obtención de la expresión regular.....	9
3. Test.....	9
3. Práctica 3.....	10
1. Presentación del problema.....	10
2. Problema escogido.....	10
2.1. Presentación del problema.....	10
2.2. Representación mediante un AFD.....	10
3. Solución usando un fichero lex.....	11
4. Bibliografía.....	12
4. Práctica 4.....	13
1. Presentación del problema.....	13
2. Problema escogido.....	13
3. Comprobando la ambigüedad de la cadena.....	13
3.1. Planteamiento de ambigüedad.....	13
3.2. Obtención con JFLAP.....	14
4. Comparativa de la obtención.....	16

1. Práctica 1

1. Presentación del problema

Enunciado: Crear un AFD (autómata finito determinístico) que acepte cadenas de ceros y de unos cuyo número de ceros no sea múltiplo de 3. De esta manera el autómata debe:

- Aceptar: 011, 1100, 011000
- Rechazar: ϵ , 000, 10100000

2. Primera aproximación

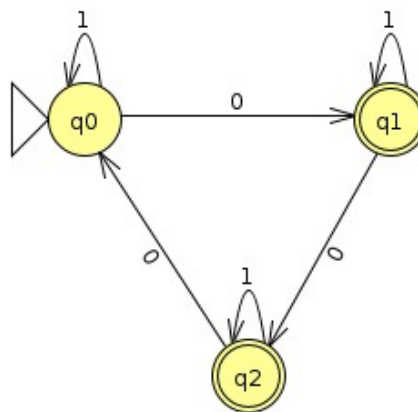


Figura 1

Una primera aproximación para representar el autómata es el que se muestra en la figura con tres estados (q_0 , q_1 , q_2). Como el número de unos introducidos no es relevante, el autómata realiza una transición hacia el mismo estado al leer un 1, mientras que al leer un 0 el autómata pasa al siguiente estado. El estado inicial es q_0 y los estados finales son q_1 y q_2 , por lo que cuando lee un número de ceros múltiplo de 3 vuelve a q_0 y en otro caso está en q_1 o q_2 .

3. Obtención del autómata a partir de una expresión regular

3.1. Problema del autómata inicial

El problema del autómata anterior es que tiene dos estados finales. Vamos a intentar optimizarlo obteniendo un autómata finito a partir de la expresión regular generada por el primer autómata que presente solo un estado final.

3.2. Obtención de la expresión regular

Sabemos que dado un autómata finito existe siempre una expresión regular que representa el lenguaje aceptado por el autómata (demostrado en el tema 2).

La expresión regular se puede obtener representando el conjunto de cadenas del alfabeto que permiten pasar del estado q_0 a los estados q_1 y q_2 o bien utilizando el método de eliminación de estados. En nuestro caso vamos a utilizar el segundo método. Para obtener la expresión regular obtenemos las expresiones regulares para cada uno de los dos estados finales.

3.2.1. Obtención de la expresión regular r_1

Obtenemos la expresión regular r_1 para el estado final q_1 eliminando q_2 .

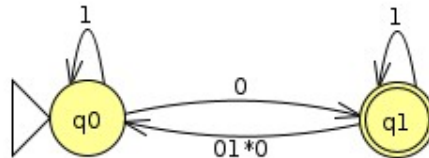


Figura 2

La expresión regular que describe el diagrama de transición es

$$r_1 = (1 + 01^*01^*0)^*01^*$$

3.2.2. Obtención de la expresión regular r_2

Obtenemos la expresión regular r_2 para el estado final q_2 eliminando q_1 .

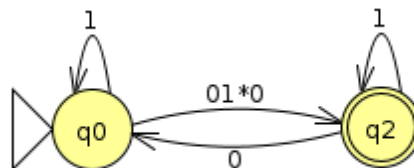


Figura 3

La expresión regular que describe el diagrama de transición es

$$r_2 = (1 + 01^*01^*0)^*01^*01^*$$

3.2.3. Obtención de la expresión regular r

La expresión regular r se puede obtener mediante la combinación de las dos expresiones anteriores de manera que

$$r = r_1 + r_2 = [(1 + 01^*01^*0)^*01^*] + [(1 + 01^*01^*0)^*01^*01^*]$$

3.2. Obtención del autómata finito no determinístico a partir de la expresión regular

En JFLAP seleccionamos la opción Regular Expression. Introducimos la expresión r y seleccionamos Convert>Convert to NFA (esto convierte la expresión regular en un autómata finito no determinístico con transiciones nulas).

El autómata que obtenemos lo convertimos a un autómata finito determinístico con la opción Convert>Convert to DFA.

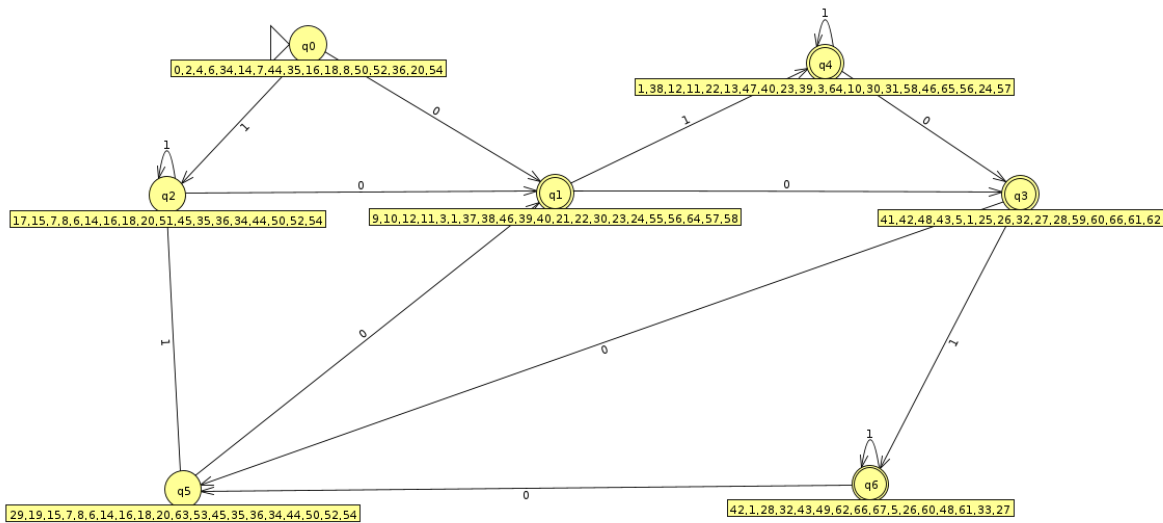


Figura 4

El autómata determinístico obtenido lo podemos minimizar en Convert>Minimize DFA. El resultado de minimizarlo es el siguiente

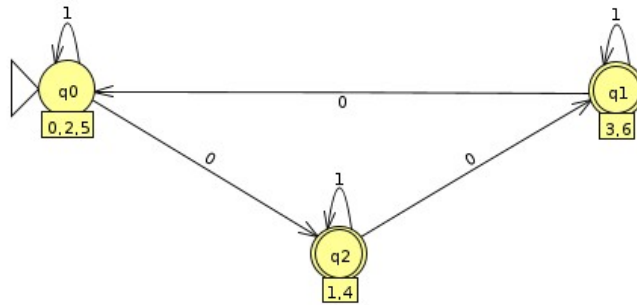


Figura 5

Como podemos observar, el diagrama de transición final obtenido es igual al diagrama de la figura 1, por lo que es un diagrama de estados óptimo para resolver el problema.

4. Test

4.1. Ejecución múltiple

Para probar el AFD se pueden utilizar las cadenas vistas en la presentación del problema. Se crea una ejecución múltiple para comprobar su correcto funcionamiento.

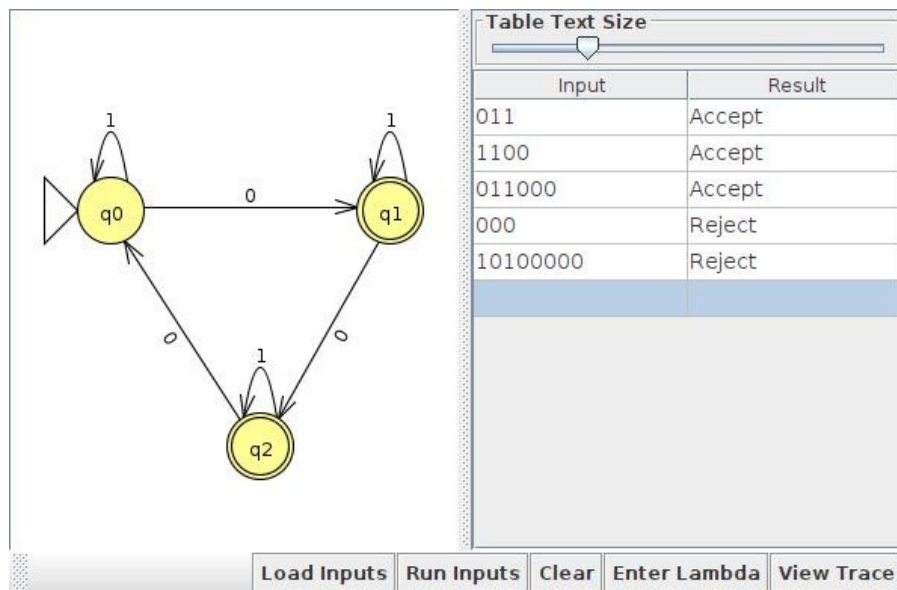


Figura 6

4.2. Ejecución paso a paso

Para la simulación se escoge el caso 1100.

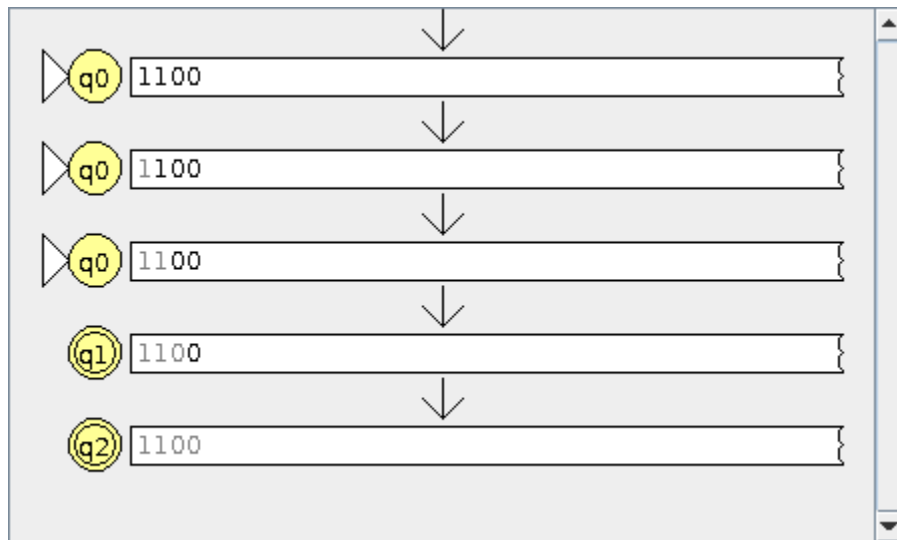


Figura 7

Para este caso, el autómata comienza desde q_0 , lee un 1 dos veces y permanece en q_0 , lee un 0 y hace una transición hacia q_1 , desde q_1 lee un 0 y hace una transición hacia q_2 y termina la cadena. Al ser q_2 un estado final, la cadena es válida.

2. Práctica 2

1. Presentación del problema

Enunciado: Elegir un lenguaje regular, representar sus cadenas mediante una expresión regular y obtener el autómata (y minimizarlos si queremos).

2. Solución

2.1. Elección del lenguaje regular

Utilizando el alfabeto $\{0,1\}$ construimos un lenguaje que permita representar las cadenas en las que el número de ceros es par y el número de unos es impar. De esta manera el autómata debe:

- Aceptar: 001, 100, 010, 0011010
- Rechazar: ϵ , 01, 101, 01100

2.2. Obtención del autómata

El autómata se puede ver como la combinación de dos autómatas más simples:

1. Un autómata cuyo número de ceros es par.
2. Un autómata cuyo número de unos es impar.

2.2.1. Representación del autómata 1

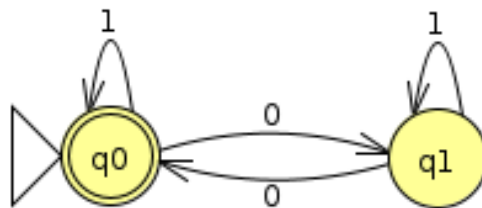


Figura 1

Para este autómata el número de unos no es importante, por lo que cuando se lee un uno se hacen transiciones hacia el mismo estado y cuando se lee un cero se cambia de estado.

2.2.2. Representación del autómata 2

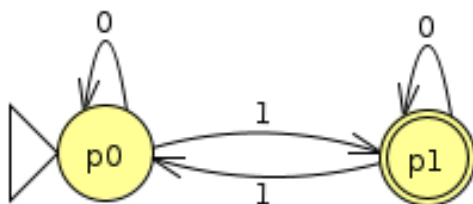


Figura 2

Para este autómata el número de ceros no es importante, por lo que cuando se lee un cero se hacen transiciones hacia el mismo estado y cuando se lee un uno se cambia de estado.

2.3. Representación del autómata final

Cada uno de los estados del autómata final representan un conjunto de estados formado por los valores de los estados de cada uno de los autómatas que lo forman. Así el estado q0 representa los valores [q0,p0], el estado q1 [q1,p0], el estado q2 [q0,p1] y el estado q3 [q1,p1]. Los dos autómatas anteriores los podemos combinar con JFLAP con la opción Convert>Combine automata y nos devuelve como resultado el autómata final.

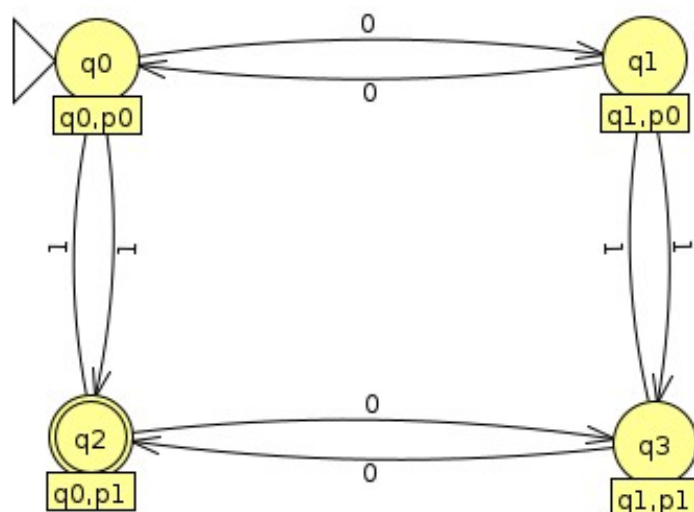


Figura 3

Intentamos minimizar el autómata obtenido con la opción Convert>Minimize DFA, pero el resultado final es el mismo; es decir, el autómata final obtenido es minimal.

2.4. Obtención de la expresión regular

Para obtener la expresión regular asociada al autómata desde JFLAP encontramos la opción Convert>Convert FA to RE. La expresión regular obtenida es

$$r = ((00+01(11)^*10)^*(1+01(11)^*0)(0(11)^*0)^*(1+0(11)^*10))^*(00+01(11)^*10)^*(1+01(11)^*0)(0(11)^*0)^*$$

Como se puede ver, para este problema es más fácil obtener la expresión regular partiendo del autómata que si se intentara hacer al contrario.

3. Test

Para probar el autómata vamos a crear una ejecución múltiple con las cadenas del apartado 2.1.

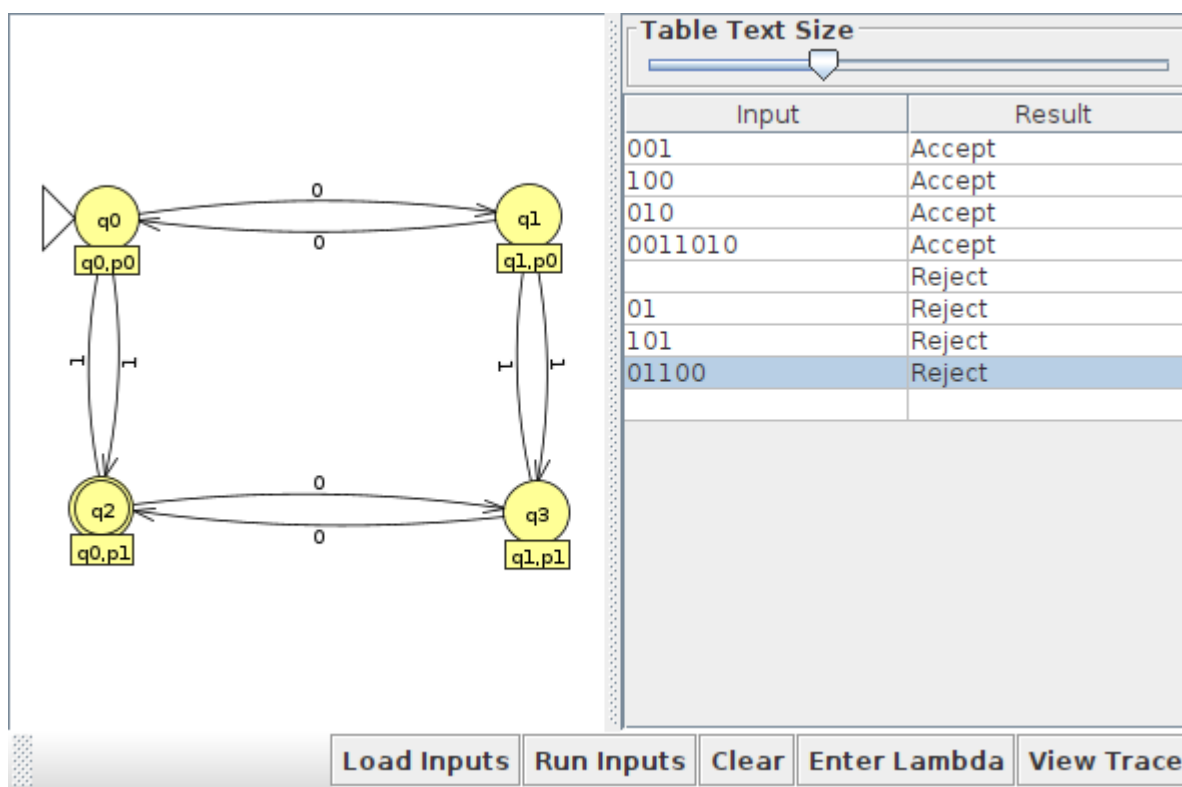


Figura 4

3. Práctica 3

1. Presentación del problema

Enunciado: Analizar un lenguaje regular con un LEX. El requisito mínimo es hacer lo mismo que hay en las diapositivas de teoría.

2. Problema escogido

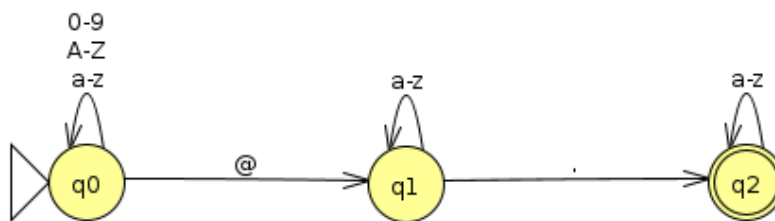
2.1. Presentación del problema

Construimos un lenguaje que permita validar una dirección de correo electrónico. Para ello la cadena debe seguir el siguiente orden:

1. El nombre del usuario (Ej.: ricksanchez)
2. El caracter “@”
3. El nombre del dominio (Ej.: ugr.es)

2.2. Representación mediante un AFD

Para expresarlo de una manera más gráfica, utilizamos un AFD.



Solución mediante un AFD

3. Solución usando un fichero lex



```
1 %%  
2  
3 [a-zA-Z0-9]{2,}@[a-z]{2,}(\.[a-z]{2,})*| {printf("\nDireccion valida\n");}  
4  
5 .+ {printf("\nDireccion incorrecta\n");}  
6 %%  
7 int main() {  
8  
9     yylex();  
10    printf("\n");  
11  
12    return 0;  
13 }
```

Código para validar un correo

Si el correo cumple con el orden establecido en el apartado 2 se imprime por pantalla “Direccion valida”; en otro caso (mostrado como .+) se imprime “Direccion incorrecta”.

Nuestro fichero cumple con el siguiente formato:

definiciones

%%

reglas

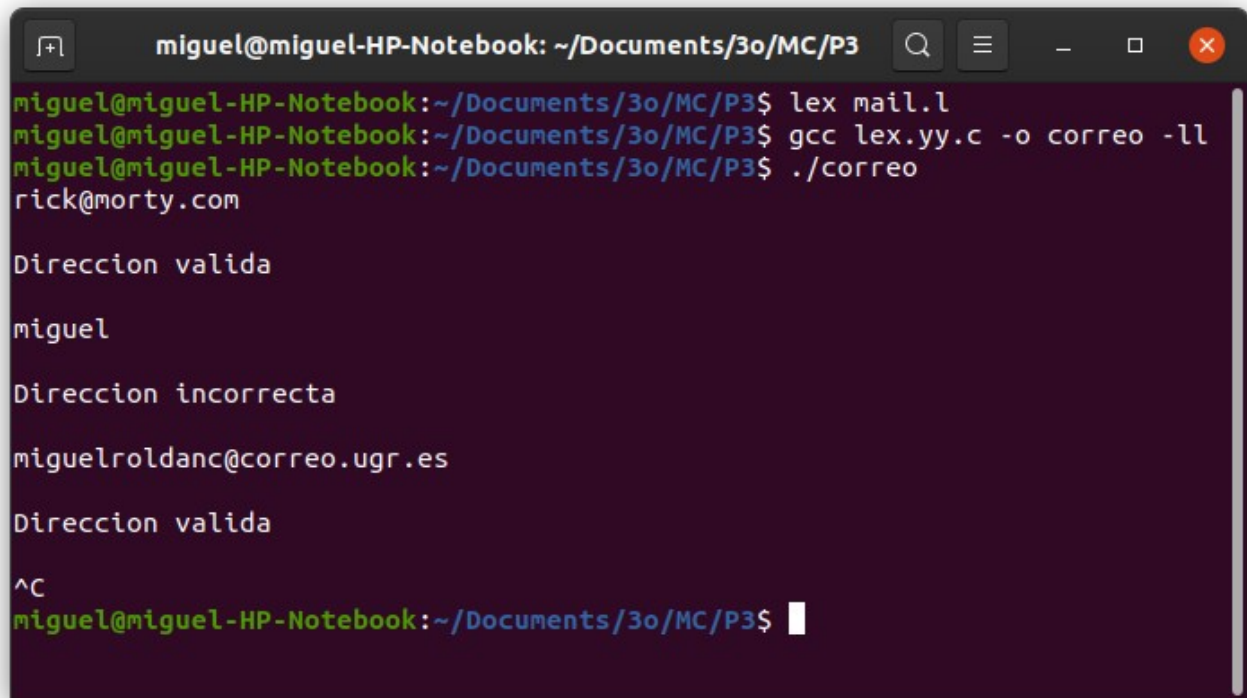
%%

subrutinas del usuario

Analicemos cada uno de los bloques para comprender el código:

- El bloque *definiciones* está vacío, y por tanto el documento empieza por los delimitadores %%. Estos demilitadores deben ponerse siempre antes de las reglas
- El bloque *reglas* está compuesto por dos reglas. La primera mostrada en la línea 3 indica que se debe imprimir “Direccion valida” cuando se lea la cadena que hay a su izquierda. La segunda mostrada en la línea 5 indica que se debe imprimir “Direccion incorrecta” en otro caso diferente a la primera regla.

- El bloque *subrutinas del usuario* está compuesto por el main. En el main se llama a la función yylex() que reconoce las expresiones regulares y ejecuta las acciones. Ésta función trabaja de manera infinita.



```
miguel@miguel-HP-Notebook: ~/Documents/3o/MC/P3
miguel@miguel-HP-Notebook:~/Documents/3o/MC/P3$ lex mail.l
miguel@miguel-HP-Notebook:~/Documents/3o/MC/P3$ gcc lex.yy.c -o correo -ll
miguel@miguel-HP-Notebook:~/Documents/3o/MC/P3$ ./correo
rick@morty.com

Direccion valida

miguel

Direccion incorrecta

miguelroldanc@correo.ugr.es

Direccion valida

^C
miguel@miguel-HP-Notebook:~/Documents/3o/MC/P3$
```

Prueba de ejecución

Como podemos observar admite tanto dominios simples en el primer caso como dominios compuestos en el tercer caso. Además al recibir una cadena sin la estructura esperada, la rechaza.

4. Bibliografía

- How to use lex command:
https://www.ibm.com/support/knowledgecenter/ssw_aix_72/l_commands/lex.html

4. Práctica 4

1. Presentación del problema

Enunciado: Coger una gramática libre de contexto ambigua y usar JFLAP para comprobar que es ambigua.

2. Problema escogido

Para el caso se escoge una gramática con las siguientes producciones:

$$E \rightarrow I \qquad E \rightarrow I - E \qquad E \rightarrow E - I \qquad I \rightarrow a|b|c|d$$

3. Comprobando la ambigüedad de la cadena

3.1. Planteamiento de ambigüedad

Para comprobar que es una gramática ambigua se debe encontrar una palabra que admita dos árboles de derivación distintos. Para ello se aplica derivación por la derecha para un árbol y derivación por la izquierda para el otro.

La palabra escogida es:

$$a-b-c$$

3.2. Obtención con JFLAP

3.2.1. Obtención del árbol de derivación por la izquierda

Start Pause Step Noninverted Tree

Input: a-b-c
String accepted! 31 nodes generated.

Input Field Text Size (For optimization, move one of the window size adjustor...)

Table Text Size

LHS		RHS
E	→	I
E	→	I-E
E	→	E-I
I	→	a
I	→	b
I	→	c
I	→	d

Start Pause Step Derivation Table

Input: a-b-c
String accepted! 31 nodes generated.

Input Field Text Size (For optimization, move one of the window size adjustor...)

Table Text Size

LHS		RHS
E	→	I
E	→	I-E
E	→	E-I
I	→	a
I	→	b
I	→	c
I	→	d

Table Text Size	
E	E
E → I-E	I-E
E → I-E	I-I-E
E → I	I-I-I
I → a	a-I-I
I → b	a-b-I
I → c	a-b-c

Derived c from I. Derivations complete.

3.2.1. Obtención del árbol de derivación por la derecha

Start Pause Step Noninverted Tree

Input: a-b-c
String accepted! 31 nodes generated.

Input Field Text Size (For optimization, move one of the window size adjustor...)

Table Text Size

LHS	RHS
E	→ I
E	→ E-I
E	→ I-E
I	→ a
I	→ b
I	→ c
I	→ d

Derived c from I. Derivations complete.

Start Pause Step Derivation Table

Input: a-b-c
String accepted! 31 nodes generated.

Input Field Text Size (For optimization, move one of the window size adjustor...)

Table Text Size

LHS	RHS
E	→ I
E	→ E-I
E	→ I-E
I	→ a
I	→ b
I	→ c
I	→ d

Table Text Size	
E	E
E → E-I	E-I
E → E-I	E-I-I
E → I	I-I-I
I → a	a-I-I
I → b	a-b-I
I → c	a-b-c

Derived c from I. Derivations complete.

4. Comparativa de la obtención

Ambos árboles de derivación son obtenidos mediante el Brute Parser. Este método ejecuta las reglas secuencialmente conforme se encuentran en la tabla.

Para obtener el árbol de derivación por la izquierda se aplica primero la regla $E \rightarrow I - E$ mientras que para obtener el árbol de derivación por la derecha se aplica primero la regla $E \rightarrow E - I$.

De esta manera en la tabla de la derivación a la izquierda se aplica consecutivamente la regla $E \rightarrow I - E$ mientras que en la tabla de la derivación a la derecha se aplica consecutivamente la regla $E \rightarrow E - I$.

Por tanto la palabra a-b-c admite dos árboles de derivación distintos y la gramática es ambigua. Esta gramática dejaría de ser ambigua si se eliminase la regla $E \rightarrow I - E$ o la regla $E \rightarrow E - I$ (solo una de ambas).