

MODELOS DE COMPUTACIÓN (2016-2017)
GRADO EN INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE GRANADA

Prácticas

David Criado Ramón

Índice

1	Demostrar que la siguiente gramática genera todas las cadenas formadas por a y b con mismo número de símbolos terminales a que de símbolos terminales b.	3
1.1	Demostrar que generamos sólo cadenas que verifican la propiedad dada. . .	3
1.2	Demostrar que generamos todas las cadenas que verifican la propiedad dada. .	4
2	Determinar si la siguiente gramática genera un lenguaje regular (tipo 3).	4
2.1	Determinar el lenguaje que es generado la gramática	4
2.2	¿Podemos encontrar una gramática regular que genere el lenguaje?	5
3	Hacer la máquina de Mealy que decodifica las cadenas generadas por la siguiente máquina:	5
4	Ejemplo de uso de LEX	7
5	Estudiar la ambigüedad generada por la siguiente gramática:	9
5.1	¿Es la gramática ambigua?	9
5.2	¿Es el lenguaje inherentemente ambiguo?	10
5.3	¿Existe una gramática que no sea ambigua y genera el mismo lenguaje? .	10

1. Demostrar que la siguiente gramática genera todas las cadenas formadas por a y b con mismo número de símbolos terminales a que de símbolos terminales b.

$G = (V, T, P, S)$, $V = \{S, A, B\}$, $T = \{a, b\}$, donde P es el conjunto de las siguientes reglas de producción:

$$\begin{array}{llll} S \rightarrow aB & S \rightarrow bA & A \rightarrow a & A \rightarrow aS \\ A \rightarrow bAA & B \rightarrow b & B \rightarrow bS & B \rightarrow aBB \end{array}$$

Para demostrarlo dividimos el proceso en dos partes:

1.1. Demostrar que generamos sólo cadenas que verifican la propiedad dada.

Para resolverlo vamos a intentar ver la interpretación que cada una de las variables de la gramática. Puesto que el objetivo es obtener cadenas que tengan el mismo número de letras a que de letras b, suponemos que esta será la interpretación de el símbolo inicial S.

Si observamos las producciones que parten de la variable A, vemos que podemos o cambiarlo por el símbolo terminal a, por un símbolo terminal a junto a la variable inicial S, o por un símbolo terminal b junto a dos variables A. Si nos fijamos, por ejemplo, en el símbolo terminal a junto a la variable inicial S observamos que si seguimos la definición que debe de ocurrir para que la propiedad ocurra una variable A genera una cadena que tiene exactamente un símbolo terminal a más que el número de símbolos terminales b. Ahora comprobemos que esta supuesta interpretación es correcta en los otros dos casos, en el que la variable A pasa al símbolo terminal a es evidente que hay un símbolo más a (1) que b (0), y en el que la variable A pasa al símbolo b junto a dos variables A, debido a la interpretación que hemos dado sabemos que las variables A producirán siempre una a más que b, así pues, al añadir dos variables A en la parte derecha es necesario que también aparezca el símbolo terminal b para cumplir la interpretación.

En el caso de la variable B, podemos observar que las producciones son idénticas a las de la variable A intercambiando los símbolos terminales y cambiando las variables A por B.

Puesto que del símbolo inicial sólo podemos generar una cadena que empiece por a y símbolo terminal B (que garantizará que cuando se desarrolle la B habrá un símbolo terminal b más que a), o ir del símbolo inicial al caso análogo (empezar por b seguida de símbolo terminal A) podemos verificar que se generan sólo las cadenas deseadas.

1.2. Demostrar que generamos todas las cadenas que verifican la propiedad dada.

Para demostrarlo hemos de ver todas las líneas de derivación posibles que pueden generar esta gramática y ver que efectivamente es posibles generarlas todas. A partir del símbolo inicial S siempre podemos generar o una la cadena aB o la cadena Ba . Si consideramos la cadena aB , para poder generar todas las cosas a partir de la variable B necesitamos:

- Acabar con una única b para equilibrar el número de a y de b disponibles ($S \rightarrow aB$, $B \rightarrow b$, ab).
- Añadir la b y dar posibilidad de continuar con una a ($S \rightarrow aB$, $B \rightarrow bS$, abS , $S \rightarrow aB$, $abaB$, etc...).
- Añadir una b y dar posibilidad de continuar con otra b ($S \rightarrow aB$, $B \rightarrow bS$, abS , $S \rightarrow bA$, $abbA$, etc...).
- Añadir una a y poder continuar con otra a ($S \rightarrow aB$, $B \rightarrow aBB$, $aaBB$, $B \rightarrow aBB$, $aaaBBB$, etc...).
- Añadir una a y poder continuar con una b ($S \rightarrow aB$, $B \rightarrow aBB$, $aaBB$ $B \rightarrow b$, $aabB$, etc...).

Puesto que los casos a partir de la variable B , son análogos a partir de la variable A , podemos afirmar que se pueden generar todas las cadenas con el criterio dado.

2. Determinar si la siguiente gramática genera un lenguaje regular (tipo 3).

Gramática libre del contexto $G = (V, T, P, S)$, $V = \{S, A, B\}$, $T = \{a, b, c, d\}$, donde P es el conjunto de las siguientes reglas de producción:

$$\begin{array}{lll} S \rightarrow AB & A \rightarrow Ab & A \rightarrow a \\ B \rightarrow cB & B \rightarrow d & \end{array}$$

Para determinar si la gramática genera un lenguaje regular dividimos el proceso en dos partes:

2.1. Determinar el lenguaje que es generado la gramática

Para ello miramos lo podemos obtener de la líneas de derivación. Partiendo del símbolo inicial S sólo podemos llegar a la cadena AB .

Partiendo de la variable A , podemos o mantener la variable y añadir un símbolo terminal b todas las veces que queramos (repitiendo la misma producción) o pasar la variable A a un único símbolo terminal a que se encontrará a la izquierda. Por tanto, la cadena

empieza con un único símbolo terminal a y una cantidad indeterminada (se admite 0), i , de letras b .

Partiendo de la variable B , podemos o añadir un número indeterminado, j , de letras c y mantener la variable (repitiendo la misma producción) o pasar la variable B a un única d , por tanto, podemos determinar que el lenguaje generador por la gramática proporcionada es $\{L = ab^i c^j d : i, j \in \mathbb{N}\}$

2.2. ¿Podemos encontrar una gramática regular que genere el lenguaje?

Puesto que es necesario que la cadena inicial podemos añadir una producción que nos permita añadir la a y una variable para continuar con el resto de la cadena, por ejemplo, $S \rightarrow aB$. La B nos ha de permitir continuar con una serie de símbolos terminales opcionales ($B \rightarrow bB$) o pasar a otra variable C ($B \rightarrow C$) que se encargará de añadir todas los símbolos terminales c que queramos ($C \rightarrow cC$) o incluso 0 si no usamos esta producción y usamos sólo la que nos permite añadir una d al final a partir de una variable C ($C \rightarrow d$). Por tanto, hemos podido encontrar una gramática regular que genera el mismo lenguaje.

3. Hacer la máquina de Mealy que decodifica las cadenas generadas por la siguiente máquina:

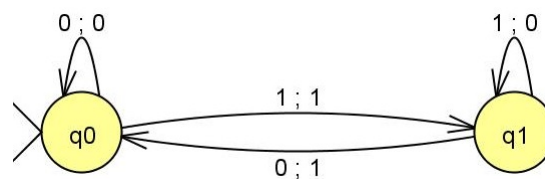


Figura 3.1: Máquina de Mealy para la codificación.

Para entender cómo la máquina codifica el mensaje hemos de darle una interpretación y ver qué ocurre en cada uno de los casos. Podemos observar que desde el estado $q0$ ocurre o que bien acabamos de empezar la lectura de la cadena o que lo que hemos leído previamente es un 0. El estado $q1$ queda determinado por haber leído previamente un 1. Por tanto, para facilitar la visualización de las transiciones voy a organizar los datos en una tabla.

Primer símbolo $q(0)$	0 \rightarrow 0
	1 \rightarrow 1
Anteriormente leído 0 ($q0$)	0 \rightarrow 0
	1 \rightarrow 1
Anteriormente leído 1 ($q1$)	0 \rightarrow 1
	1 \rightarrow 0

Tabla 3.1: Tabla de transiciones (*leído \rightarrow escrito*) para la máquina de la codificación.

Para hacer la tabla que decodifique simplemente hemos de invertir cada una de las transiciones de la tabla original, obteniendo por tanto:

Primer símbolo $q(0)$	0 \rightarrow 0
	1 \rightarrow 1
Anteriormente leído 0 ($q0$)	0 \rightarrow 0
	1 \rightarrow 1
Anteriormente leído 1 ($q1$)	1 \rightarrow 0
	0 \rightarrow 1

Tabla 3.2: Tabla de transiciones (*leído \rightarrow escrito*) para la máquina de la decodificación.

Y siguiendo la representación gráfica del autómata la siguiente:

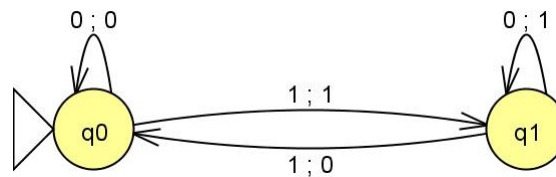


Figura 3.2: Máquina de Mealy para la decodificación.

Para comprobar que funciona correctamente vamos a comprobarlo usando JFLAP. Para ello introducimos la cadena de ejemplo 1000101 en la máquina de Mealy que codifica el mensaje y la codificación obtenida la pasamos por la máquina que decodifica tal y como podemos comprobar en la siguientes capturas:

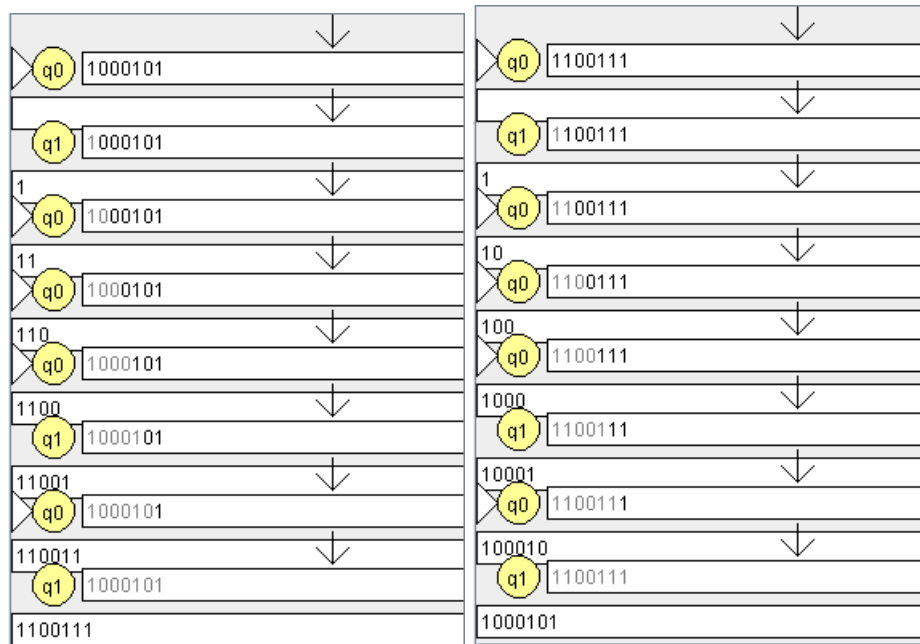
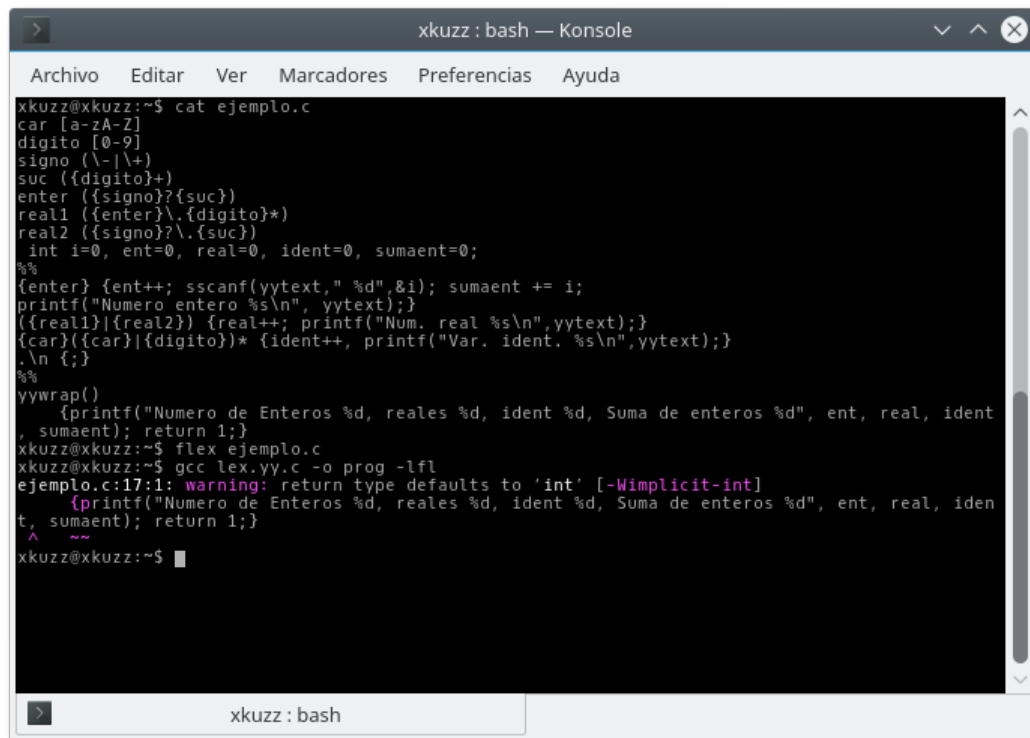


Figura 3.3: Procesos de codificación y decodificación de la cadena ejemplo.

4. Ejemplo de uso de LEX

En esta práctica vamos a probar el uso de lex, en concreto voy a usar flex (fast lex). Para instalarlo sobre kubuntu 16.10 utilizamos el comando `sudo apt-get install flex`.

Vamos a usar el archivo de ejemplo que vemos en el cat de la imagen. En el definimos las expresiones regulares para identificar: conjuntos de letras (no valen letras con tilde), dígitos, signos, sucesiones de dígitos, enteros (combinación de dígitos que pueden tener un signo al principio) y reales (que pueden empezar con signo o no y han de tener una sucesión de dígitos seguida de un punto y otra sucesión de dígitos). Cada vez que se encuentra una ocurrencia de los mismos u combinaciones de ellos es sacada por la salida estándar (como podemos observar en la segunda sección del archivo) y en concreto se suman los enteros para mostrar las ocurrencias de cada tipo (como podemos ver en la tercera sección).

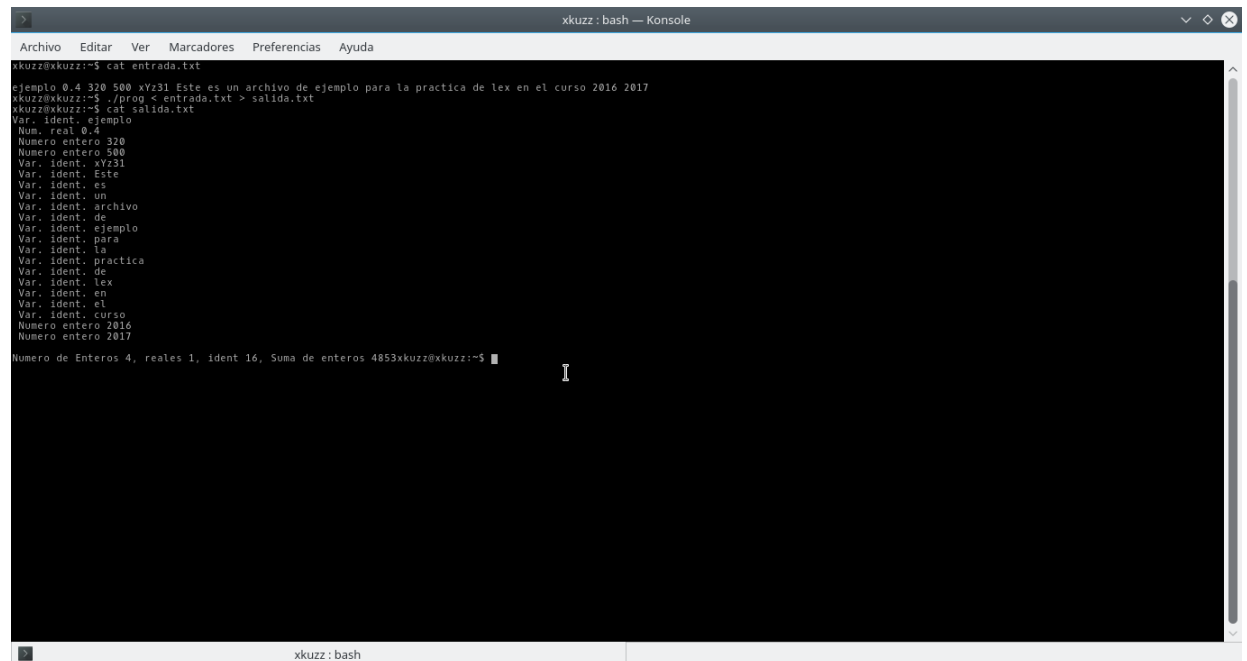


```

xkuzz : bash — Konsole
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda
xkuzz@xkuzz:~$ cat ejemplo.c
car [a-zA-Z]
digito [0-9]
signo (\-|\+)
suc ({digito}+)
enter ({signo}?{suc})
real1 ({enter}\.{digito}*)
real2 ({signo}?\.{suc})
int i=0, ent=0, real=0, ident=0, sumaent=0;
%%
{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;}
xkuzz@xkuzz:~$ flex ejemplo.c
xkuzz@xkuzz:~$ gcc lex.yy.c -o prog -lfl
ejemplo.c:17:1: warning: return type defaults to 'int' [-Wimplicit-int]
{printf("Numero de Enteros %d, reales %d, ident %d, Suma de enteros %d", ent, real, iden
t, sumaent); return 1;}
^
~
xkuzz@xkuzz:~$
```

Figura 4.1: Archivo de ejemplo y comandos de compilación.

Para empezar el proceso de compilación ejecutamos el comando flex sobre el archivo en el que hemos escrito las expresiones regulares. La ejecución de flex dará lugar a la creación de un archivo lex.yy.c que será un ejecutable en C y para compilarlo utilizamos gcc linkando con la librería, es decir, utilizamos el comando `gcc lex.yy.c -o prog -lfl`. Por tanto ya sólo nos queda utilizar el programa compilado y redireccionando la entrada y salida estándar a los archivos que deseemos.



```
xkuzz: bash - Konsole
Archivo Editar Ver Marcadores Preferencias Ayuda
xkuzz@xkuzz:~$ cat entrada.txt
ejemplo 0.4 320 500 xyz31 Este es un archivo de ejemplo para la practica de lex en el curso 2016 2017
xkuzz@xkuzz:~$ ./prog < entrada.txt > salida.txt
xkuzz@xkuzz:~$ cat salida.txt
Var. ident. ejemplo
Num. real 0.4
Numero entero 320
Numero entero 500
Var. ident. xyz31
Var. ident. Este
Var. ident. es
Var. ident. un
Var. ident. archivo
Var. ident. de
Var. ident. ejemplo
Var. ident. para
Var. ident. la
Var. ident. practica
Var. ident. de
Var. ident. lex
Var. ident. en
Var. ident. el
Var. ident. curso
Numero entero 2016
Numero entero 2017
Numero de Enteros 4, reales 1, ident 16, Suma de enteros 4853xkuzz:xkuzz:~$
```

Figura 4.2: Uso del programa compilado para analizar un archivo de prueba.

Si vemos el contenido del archivo de salida vemos como se muestra cada ocurrencia de cada tipo de expresiones regulares y se han contabilizado cuántas habían salido al ejecutar el programa.

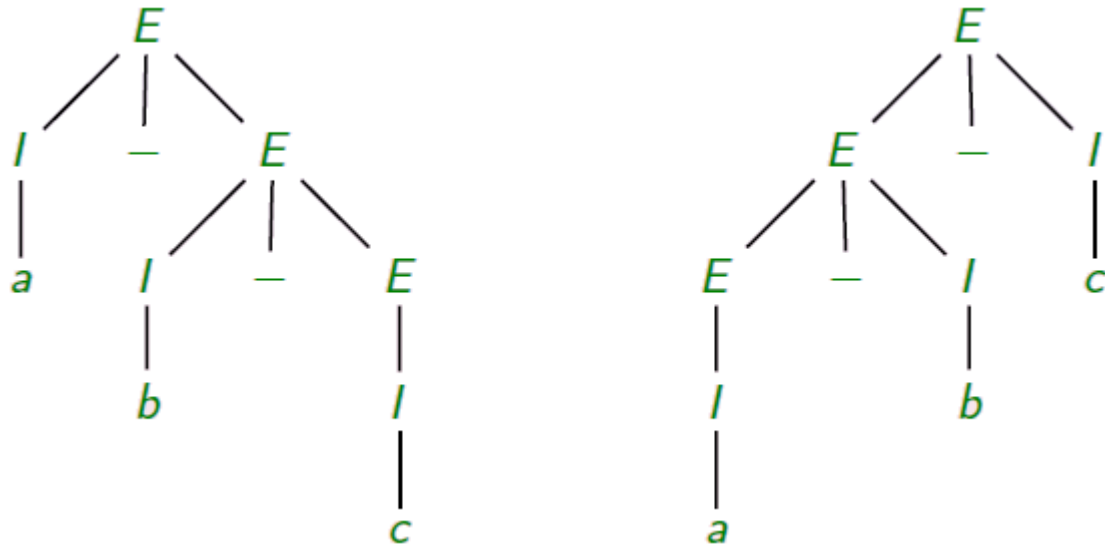
5. Estudiar la ambigüedad generada por la siguiente gramática:

$G = (V, T, P, E)$, $V = \{E, I\}$, $T = \{a, b, c, d\}$, donde P es el conjunto de las siguientes reglas de producción:

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

5.1. ¿Es la gramática ambigua?

Una gramática es ambigua si para al menos una cadena generada por la gramática puede ser obtenida mediante dos árboles de derivación distintas. Así pues la manera más simple de demostrar que una gramática es ambigua es encontrar dicha cadena que es generada por varios árboles de derivación. En nuestro caso es bastante fácil ver que $a - b - c$ es una cadena generada por los dos siguientes árboles de derivación.



5.2. ¿Es el lenguaje inherentemente ambiguo?

Un lenguaje inherentemente ambiguo es aquel para el que toda gramática es ambigua. Por tanto, igual que antes es fácil dar un contraejemplo encontrando una gramática que no sea ambigua y genere el mismo lenguaje.

5.3. ¿Existe una gramática que no sea ambigua y genera el mismo lenguaje?

La clave para resolver este problema es que puesto que tenemos la producción $E \rightarrow I - E$ y su inversa $E \rightarrow E - I$ y la existencia de una producción que nos permite transformar la E en I , hace que tengamos dos formas distintas de obtener variables del tipo $I - I$. Por tanto, para solucionar los problemas de ambigüedad generados por la gramática o bien quitamos la producción $E \rightarrow I - E$ o bien la otra producción: $E \rightarrow E - I$ y dejando el resto de producciones intactas.