Práctica 2
Manejo Avanzado de *Flex*

Jorge Bernad, Elvira Mayordomo, Mónica Hernández, José Manuel Colom

Tareas

1. Estudia la sección sobre las *condiciones de arranque* en el libro *flex & bison* (páginas 28 a 31) y el capítulo 9 del manual de *Flex* disponible en moodle.

2. Lee la introducción de esta práctica y realiza los ejercicios propuestos.

3. Elabora la memoria de la práctica y entrégala junto con los ficheros fuente según el Procedimiento de Entrega de Prácticas explicado en la Introducción a las Prácticas de la Asignatura. La fecha tope de entrega será hasta el día anterior al comienzo de la Práctica 3.

Nota: El incumplimiento de las normas de entrega se reflejará en la calificación de la práctica.

Se recuerda especialmente lo siguiente:

- Crea un directorio que contenga exclusivamente el fichero con la memoria en formato *PDF*, los ficheros fuente con tu código *.l* de *Flex*, y los de prueba (*.txt* de texto). No usar subdirectorios.

- Accede al directorio con tus ficheros y ejecuta el comando

```
zip nipPr2.zip *.*
```

  donde *nip* es el identificador personal.

- En caso de que el fichero resultante tenga un tamaño mayor de 1024 KB deberás dividir el fichero `nipPr2.zip` en varios ficheros de tamaño 1024KB con el comando de linux

```
split -b 1m nipPr2.zip nipPr2.zip.
```

Introducción

El objetivo principal de esta práctica es aprender a desarrollar analizadores léxicos en *Flex* más sofisticados, profundizando en el manejo de lo que se conoce como *condiciones de arranque*. Las *condiciones de arranque* no son imprescindibles, ya que siempre se pueden emular utilizando código *C* en las acciones de los patrones. No obstante, su uso facilita mucho el desarrollo de los programas en *Flex,* ya que ayudan a estructurar conjuntos de patrones/acciones en función de un contexto determinado o condiciones previas.

Ejercicio 1

GenBank (`https://www.ncbi.nlm.nih.gov/genbank/`) es una base de datos que almacena la información genética de todas las secuencias de ADN públicas existentes. El formato de ficheros más utilizado por GenBank para almacenar esta información es el formato de ficheros de texto FASTA. El formato FASTA consiste en dos partes diferenciadas. La primera es la cabecera del fichero, que comienza por el símbolo $>$ seguida del identificador de secuencia y un texto que hace referencia a los detalles de la cadena de ADN almacenada en el fichero. Después del final de línea comienza la segunda parte del fichero, en las que se encuentra la secuencia correspondiente de nucleótidos, denotados por los caracteres ACTG. En esta práctica se trabajará con ficheros FASTA que puedan contener más de una secuencia.

Ejemplo:

```
>AB128931.1 Homo sapiens mRNA for hypothetic protein, complete cds,
Alzheimer disease site specific expressed gene
TTCCGGCCGAGGTACACATGGAGCCATACATAGAGTCACATTAGTGGTGGTTTGTTCTCCTAGGTGCTTG
AAGGGCTCCTGAGTGCCAAGTCATTGTATATGTAGTTACCAAGGTAAATGTAAGTAATTAAGGGAACAGA
GTGCAGTGGCTTCCACTGTGCTATTTCTCACTTTGTTTTGCATATTGCTAGGATCCAGAGCGATATGATT
GGCATTTGAGGGAAGATAAGCACCATACTTGTGTTGCTTATTTTCTTTTTAATTAAGTTGCCTTCTGACA
ATCATACTTTCTGACATTTATTTAGCCTTTACATTTTATAAATCACTTTCACATATGCTGTTTCATTTAG
>AF315290.1 Mus musculus collagen-like Alzheimer amyloid plaque component
precursor type I mRNA, complete cds
CCCGGCGCCACACAGTCCCCGGCCGGAGGGTGCTTTTCACTCCTAGCTGGAAGGGGAGAAAGAATCTGGA
GGACGGTCGGTCCACGCCTGCTGATCCGGACGCCGAGCCACGCGCAGGTCCATCTCTAAGCCCGGGCTCC
GACTCTACCAACTAGTTGTGCAGCCGCAGGGACTGAACTTTGGAGGAACCGACCCTTCCTCTCATTCTAA
GATTACTGGAGGAGATAGAAGGTGGAAGGCGTAGCGGAGGCCAGCGACCCCGCCACAATGTTGGTGAAGA
AGCTTGCAGGGAAAGGAGGGGGACGAGAGTCTGGATCAGAAGATCCGCGCCCCTTGGGACAGCGTTGTGC
CGGCACCATGCCCTCGTGCACGGCCCTGGCGACCCTCTTGTCAGTGGTTGCTGTGGCTTTCTGTTTTTA
>AF293341.1 Homo sapiens collagen-like Alzheimer amyloid plaque component
precursor type II mRNA, complete cds
CCGCGACTTCGGCTTCGCGAGTAGCATTGGTTCCTTGGGTTTATTTCGTTTTCCTCTCTCTTCTCCACCT
TAGTCGCCCCTTTCGCGCTGCGCTGTAGCGTGCTCTCACAGCCTTTTTGCCTTGAACTGAATGCAGGTGG
GAAACAGGTCGGCGTGCCGAAAGACACCGAGTAGGTAGAAATAAGGCAAACTCACAGAGGCGCAACAGGT
CCGGTCCTCCGTGGCCAGGGCGAGCCGCGGCCCCGCGTGGCGCCTCGGCCGTTGCCCTCGGACCCTGAGC
GGCCACTGTTGGGGCCCTCGAAAGAGGTGTCGGTCCTCTGGGAGTCGGAAGAGCTGTCTGGGTGGGTTTC
```

El objetivo de esta práctica es implementar analizadores léxicos en *Flex* (ficheros ej1.1.l y ej1.2.l) que permitan comprimir y descomprimir ficheros FASTA de acuerdo con el algoritmo *run length encoding*. El compresor transcribirá directamente las cabeceras de las secuencias de ADN de los ficheros originales. A continuación detectará secuencialmente las repeticiones de nucleótidos y escribirá en el fichero comprimido el nucléotido detectado y el número de repeticiones. Si sólo se detecta un nucleótido se escribirá directamente dicho nucleótido sin indicar que el número de repeticiones es 1. Así la secuencia TTCCGGCCGAGGTACACA se transcribirá como T2C2G2C2GAG2TACACA. El descompresor realizará el procedimiento inverso para así obtener el fichero FASTA original. Para realizar pruebas podéis utilizar el fichero FASTA proporcionado con el enunciado de esta práctica.

Ejercicio 2

El objetivo de esta práctica es implementar un analizador léxico en *Flex* (fichero ej2.l) que permita analizar la calidad del código fuente de un programa escrito en un lenguaje de programación CMM (C muy simple). Este lenguaje cuenta con los tipos int, double, char, bool y void, comentarios de línea //, comentarios de párrafo /*...*/, y permite el uso de procedimientos y funciones. Con las estructuras de control if ... else ..., for, while, do ... while CMM no permite la utilización de llaves, por lo que funciona con instrucciones de una sola línea.

La calidad del código fuente se medirá atendiendo al uso de comentarios, definición de constantes, número y la desviación estándar del tamaño de las funciones. Se obtendrá una puntuación del 1 al 10 atendiendo a los siguientes criterios:

- Si la cantidad de comentarios está entre el 10 y el 50 % del número de líneas de código se considerará una cantidad razonable y se sumarán 4 puntos.
- Si se definen constantes se sumará 1 punto.
- Si se define un número de funciones entre el 20 y el 30 % del número de líneas de código se considerará una cantidad razonable y se sumarán 2.5 puntos.
- Si la desviación estándar del tamaño de las funciones está por debajo de 5 líneas de código se considerará una cantidad razonable y se sumáran 2.5 puntos.

La salida del programa estará compuesta por exactamente siete líneas de texto con el formato:

```
T:<total de lineas de codigo fuente (excepto comentarios)>
C:<total de lineas de comentarios>
CONST:<total de constantes definidas>
F:<total de funciones definidas>
FM:<tamaño promedio de las funciones>
FS:<desviacion estandar de las funciones>
P:<puntuacion>
```

Ejemplo:

```
/* Comentario
 * de
 * parrafo
 */
#include <stdio.h>
const int PI = 3.1416;
// Metodo mostrar
int mostrar()
{
    printf( "Hola mundo \n" );
}
/* Metodo main */
void main()
{
    mostrar();
    if( PI > 2 * PI )
        printf( "Mala cosa \n" );
}
```

Salida:

```
T:12
C:6
CONST:1
F:2
FM:5.00
FS:1.0
P:7.5
```

Ejercicio 3

La distancia Levenshtein es una métrica muy utilizada para comparar dos cadenas de caracteres. Esta distancia es la base de muchos correctores ortográficos como el buscador de Google o de editores de texto como Word. Por ejemplo, al introducir en Google "*mogigato*", el buscador nos sugiere como búsqueda la versión correcta "*mojigato*". Aunque el algoritmo completo para hacer sugerencias es más complicado del que vamos a explicar en este ejercicio[1], la idea general consiste en, dada una palabra w y un corpus de palabras correctas, $\mathcal{C} = \{w_1, w_2, \ldots, w_n\}$, buscar palabras w_i dentro del corpus $\mathcal{C}$ que sean lo más parecidas posibles a la palabra w. Es aquí donde la distancia Levenshtein entra en juego: dos palabras son más parecidas cuanto menor sea su distancia Levenshtein.

Intuitivamente, la distancia Levenshtein entre dos palabras w y z, $d(w, z)$, es el mínimo número de inserciones, borrados y sustituciones de símbolos sobre la palabra w para conseguir la palabra z. Por ejemplo, si $w = cosla$ y $z = casa$, la distancia $d(w, z) = 2$: para transformar *cosla* en *casa*, necesitamos sustituir la o por una a y borrar la l, en total dos operaciones. Si $w = casa$ y $z = costas$, la distancia es $d(w, z) = 3$: sustituimos la primera a por una o; insertamos una t entre la s y la última a; e insertamos al final una s.

El objetivo de este ejercicio es implementar en Flex un analizador léxico para poder hallar todas las palabras del fichero *lemario_ es.txt*, adjunto al enunciado de esta práctica, que estén a distancia Levenshtein menor o igual a dos de la palabra "casa". El fichero *lemario_ es.txt* contiene unas 86.000 palabras del español, **cada palabra en un línea distinta**. Por tanto, nuestro corpus $\mathcal{C}$ de palabras correctas estará formado por las que aparecen en el fichero, y buscamos el conjunto B de palabras:

$$B = \{w \in \mathcal{C} \mid d(casa, w) \leq 2\}.$$

Más concretamente, se pide implementar en Flex un fichero llamado *ej3.l* para mostrar por pantalla **todas las lineas l de un texto cualquiera que cumplan $d(l, casa) \leq 2$.**

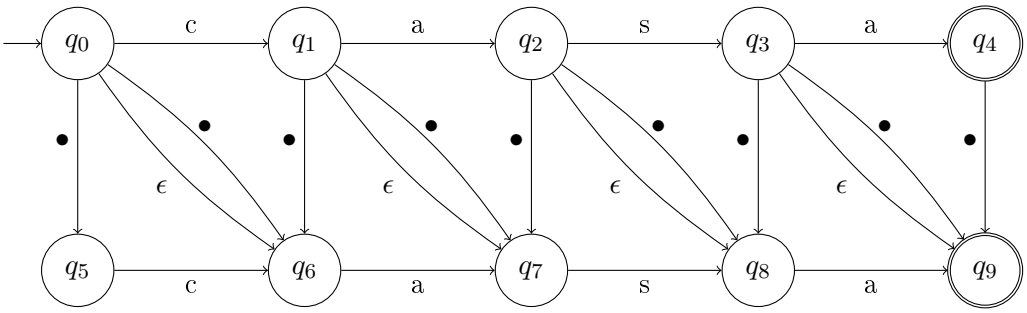
Existen varios algoritmos[2] para hallar la distancia de Levenshtein. Los más sencillos recorren varias veces los símbolos que componen las palabras teniendo, en el peor de los casos, un coste cuadrático en tiempo. El algoritmo más rápido que se conoce es en tiempo lineal, aunque la complejidad en espacio es cuadrática. Recordad que nuestro objetivo es saber si la distancia entre una palabra y "casa" es menor o igual a dos. Esto es, no necesitamos saber exactamente cuál es la distancia, solo si es mayor a dos o no. Nos planteamos si es posible diseñar un AFD para resolver el problema.

Primero vamos a ver que es fácil construir un autómata que acepte solo las palabras que estén a distancia menor o igual a 1 de *casa*. Para construir cualquier palabra a distancia 1 de *casa* podemos hacer: una inserción, como $fcasa$ o $casfa$; un borrado, asa, caa; o una sustitución, $fasa$, $cosa$.

[1] Para más detalles sobre el corrector de Google se puede consultar `http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/36180.pdf`

[2] `https://es.wikipedia.org/wiki/Distancia_de_Levenshtein`

Observemos el siguiente autómata:



Las transiciones marcadas con • son una abreviatura para indicar que hay transiciones para todo carácter distinto del salto de línea, similar a lo que significa '.' en una expresión regular de Flex. Notemos que con la primera fila de estados (estados de q_0 a q_4) se reconoce la palabra *casa*, esto es, la única palabra a distancia cero. Además,

- cualquier palabra que se consiga con una inserción se reconocerá utilizando una de las transiciones verticales. Por ejemplo, *casta* está a distancia uno de *casa* y es aceptada por el autómata con la computación: $q_0q_1q_2q_3q_8q_9$. Sin embargo, la palabra $casffa$ (dos inserciones) será rechazada por el autómata;
- cualquier palabra que se consiga mediante un borrado se reconocerá utilizando una de las ϵ-transiciones diagonales. Por ejemplo, *caa* es aceptada con la computación $q_0q_1q_2q_8q_9$;
- cualquier palabra que se consiga mediante una sustitución se reconocerá utilizando una de las transiciones diagonales marcada con •. Por ejemplo, *cosa* será aceptada mediante la computación $q_0q_1q_7q_8q_9$.

A partir de este autómata es fácil construir un AFnD que reconozca las palabras a distancia menor o igual a dos de *casa*. En la memoria se deberá añadir el autómata que se ha construido.

Como ya sabéis o estáis a punto de saber, todo AFnD es equivalente a una expresión regular. Esta expresión regular es la que podemos utilizar para implementar en Flex el analizador léxico buscado. Os aconsejamos que utilicéis la herramienta $JFLAP$[3] para hallar la expresión regular. La expresión obtenida con JFLAP la deberéis modificar para adaptarla a la sintaxis de Flex.

[3] `http://www.jflap.org/`