Práctica 2
Manejo Avanzado de *Flex*

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

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:

- Verifica que todos tus ficheros fuente (*.l*) contienen como comentario en sus primeras líneas el NIA y el nombre del autor. Todos los programas deberán estar debidamente documentados.
- Verifica que los ficheros que vas a presentar compilan y ejecutan correctamente en hendrix.
- Crea un fichero comprimido .zip llamado

  `niaPr2.zip`

  donde *nia* es el identificador personal. Este fichero comprimido **debe contener 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.
- Utiliza el enlace a una tarea de moodle disponible en la sección "Prácticas de Laboratorio" para entregar el fichero `niaPr2.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 y ARN públicas existentes. El formato de ficheros más utilizado por GenBank para almacenar esta información es el formato de ficheros de texto *FASTA*. Una secuencia en formato FASTA se compone de uno o varios bloques conteniendo:

- Una primera línea comenzada por el símbolo > seguida del identificador de la secuencia y un texto que hace referencia a los detalles de la cadena de ADN o ARN descrita.
- Tras el final de línea con la descripción, una o varias líneas con la secuencia correspondiente de nucleótidos denotados por los caracteres ACTG.

Si la molécula representada es de ARN se sobreentiende que las bases de timina (T) se corresponden con uracilo.

Ejemplo:

```
>NC_045512.2 Severe acute respiratory syndrome coronavirus 2 isolate
ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCTGTTCTCTAAA
CGAACTTTAAAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACTCACGCAGTATAATTAATAAC
TAATTACTGTCGTTGACAGGACACGAGTAACTCGTCTATCTTCTGCAGGCTGCTTACGGTTTCGTCCGTG
GGCGACGAGCTTGGCACTGATCCTTATGAAGATTTTCAAGAAAACTGGAACACTAAACATAGCAGTGGTG
TTACCCGTGAACTCATGCGTGAGCTTAACGGAGGGGCATACACTCGCTATGTCGATAACAACTTCTGTGG
CCCTGATGGCTACCCTCTTGAGTGCATTAAAGACCTTCTAGCACGTGCTGGTAAAGCTTCATGCACTTTG
TCCGAACAACTGGACTTTATTGACACTAAGAGGGGTGTATACTGCTGCCGTGAACATGAGCATGAAATTG
>MW040606.1 Severe acute respiratory syndrome coronavirus
GTGCAGAATGAATTCTCGTAACTACATAGCACAAGTAGATGTAGTTAACTTTAATCTCACATAGCAATCT
TTAATCAGTGTGTAACATTAGGGAGGACTTGAAAGAGCCACCACATTTTCACCGAGGCCACGCGGAGTAC
GATCGAGTGTACAGTGAACAATGCTAGGGAGAGCTGCCTATATGGAAGAGCCCTAATGTGTAAAATTAAT
TTTAGTAGTGCTATCCCCATGTGATTTTAATAGCTTCTTAGGAGAATGACAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA
```

El objetivo de esta práctica es implementar analizadores léxicos en *Flex* (ficheros `ej1_1.l` y `ej1_2.l`) que permitan comprimir (fichero `ej1_1.l`) y descomprimir (fichero `ej1_2.l`) ficheros FASTA de acuerdo con el algoritmo *run length encoding*. El compresor transcribirá directamente las cabeceras de las secuencias de ADN o ARN 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. Este proceso de compresión se realizará línea a línea, esto es, si una línea termina con A y la siguiente empieza también con A, en el fichero comprimido no se considerará que hay dos A consecutivas. Por tanto, el fichero comprimido contendrá tantas líneas de texto como el original. 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. Contiene la información de tres pacientes de Covid19 localizados en diferentes partes del mundo.

Ejercicio 2

La base de datos de la Alzheimers Disease Neuroimaging Initiative (ADNI) es un repositorio en el que se lleva recopilando sistemáticamente la información de un conjunto de pacientes con el objetivo de aumentar el conocimiento clínico sobre la enfermedad de Alzhéimer. A partir de esta base de datos, tenemos generados un conjunto de ficheros en formato csv, ficheros de texto con campos separados por comas, donde la primera línea contiene los nombres de los campos (columnas) del fichero. Cada línea de un fichero contiene información relativa a la visita de un paciente al especialista en un día determinado. La información del fichero relevante para nosotros es la siguiente:

- Columna `PTID`: contiene los identificadores únicos de los pacientes. Un identificador está formado por `LLL_P_XXXX`, donde cada `L` es una letra mayúscula y cada `X` un dígito. Por ejemplo, `ELQ_P_1891`.
- Columna `VISCODE`: es un código de la visita del paciente. Puede ser un valor entre: `bl`, visita baseline; una `m` seguida de una cadena formada por dígitos, indicando los meses transcurridos tras la visita baseline. Por ejemplo `m06` y `m98`, serían dos posibles valores en esta columna informando que la visita transcurrió tras 6 y 98 meses, respectivamente, desde la visita baseline.
- Columna `DX`: contiene un código con el diagnóstico del paciente en la visita. Si no se pudo obtener un diagnóstico en esa visita, el campo estará vacío (dos comas seguidas en el fichero).

Estamos interesados en realizar un estudio con los datos de los pacientes adquiridos en las visitas de baseline (`bl`), 6 meses (`m06`), 12 meses (`m12`) y 24 meses (`m24`). Un paciente

se considerará válido para nuestro estudio si posee alguna información en todas estas visitas. Para evaluar la viabilidad de nuestro estudio, nos gustaría conocer el número de pacientes total del que disponemos información, el número total de pacientes válidos para nuestro estudio y, de entre los válidos, el número de pacientes con un diagnóstico en la visita baseline de normal (`CN`), probable enfermedad de Alzhéimer (`AD`) y deterioro cognitivo leve (`*MCI`, donde `*` es una única letra mayúscula cualquiera, por ejemplo, `EMCI`, `LMCI`, etc). En el material adjunto a la práctica se puede consultar el fichero `ADNI.csv`, ejemplo de este tipo de ficheros.

Diseñar un analizador léxico que permita extraer la información deseada para evaluar la viabilidad de nuestro estudio. El fichero fuente de Flex para resolver este problema se llamará `ej2.l`. La salida del programa constará de cinco líneas de texto:

```
NP: <número total de pacientes>
NV: <número total de pacientes válidos>
CN: <número de pacientes normales entre los válidos>
AD: <número de pacientes probables de Alzhéimer entre los válidos>
MCI: <número de pacientes con deterioro cognitivo leve entre los válidos>
```

Ejemplo de salida para el fichero `ADNI.csv`:

```
NP: 2264
NV: 1229
CN: 355
AD: 174
MCI: 678
```

Consideraciones importantes sobre el formato de los ficheros de entrada:

- La información de cada paciente se encuentra en líneas consecutivas del fichero.
- Aunque en el fichero `ADNI.csv` adjunto a la práctica los campos `PTID`, `VISCODE`, y `DX`, se encuentran en las columnas 2, 3 y 6 del fichero, no siempre tiene que ser así. Lo único que podemos asegurar es que la columna `PTID` aparecerá antes que la columna `VISCODE`, y ésta, antes que `DX`. Por tanto, nos podemos encontrar un fichero, por ejemplo, con las columnas en posiciones 5 (`PTID`), 15 (`VISCODE`), 140(`DX`).
- El carácter '`,`' solo aparece para separar los campos dentro del fichero.
- No hay espacios en blanco al inicio y fin de cada valor de los campos. Dicho de otra forma, entre dos comas nunca habrá espacios en blanco. Tampoco al inicio y al final de una línea.

- Los valores bajo las tres columnas implicadas en el estudio son propios de cada columna, aunque pueden aparecer como subcadenas de los valores en otras columnas. Esto es, por ejemplo, si el valor `AD` aparece en la columna `DX`, bajo ninguna otra columna puede aparecer exactamente el valor `AD`, pero sí puede existir otra columna donde exista un valor que contiene `AD` como subcadena, como el valor `ADNI3`.

Observaciones: Al resolver este ejercicio puede ser necesario tener que hacer una copia de una cadena de caracteres en C, y comparar dos cadenas. Estas operaciones se pueden realizar con las funciones `strdup` y `strcmp` de la librería `string.h`. La función `strdup(s)` devuelve una copia de la cadena `s`, y la función `strcmp(s1, s2)` devuelve 0 si y solo si las cadenas `s1` y `s2` son iguales.

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 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:

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

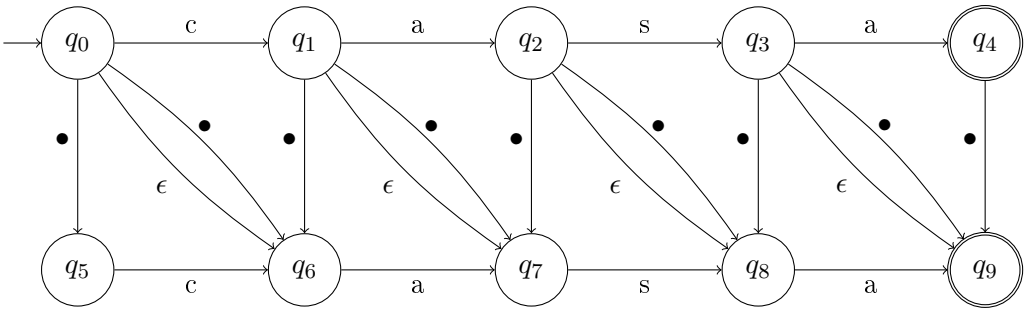
Más concretamente, se pide implementar en Flex un fichero llamado *ej3.l* para mostrar por pantalla **todas las lineas l de un texto cualquiera que cumplan $d(l, casa) \leq 2$.**

[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`

Existen varios algoritmos[2] para hallar la distancia de Levenshtein, todos ellos en tiempo cuadrático. 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*.

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.

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

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/`