Universidad de Granada

Grado en Ingeniería Informática y Matemáticas

Modelos de Computación

Práctica de Lex/Flex

Realizado por:

Miriam García Sollo
Juana María Rascón Contreras

23 de diciembre de 2025

Índice

1. Introducción

El objetivo de esta práctica de la asignatura Modelos de Computación consiste en el diseño y desarrollo de un analizador léxico utilizando la herramienta Flex, integrada junto con el lenguaje de programación C++. Veremos cómo autómatas y expresiones regulares pueden aplicarse para procesar flujos de información, transformando datos crudos en estructuras útiles para el usuario.

1.1. Motivación y Objetivos

Para el desarrollo del trabajo, hemos contado con libertad para elegir la fuente de datos, por lo que en este trabajo buscamos enfrentarnos al procesamiento de un documento web real en formato HTML.

Nuestra elección se ha centrado en los informes de Salud Pública de la Junta de Andalucía, específicamente en los datos de las Enfermedades de Declaración Obligatoria (EDO). Esta decisión se ha tomado por dos motivos.

En primer lugar, existe un interés social evidente en trabajar con datos sanitarios reales. Tratar con información sobre la incidencia de enfermedades nos permite actuar con problemas del mundo real.

En segundo lugar, vimos en este proyecto una oportunidad ideal para familiarizarnos con el lenguaje HTML. Al tener que realizar partir de una página web para poder extraer los datos, nos hemos visto obligadas a aprender cómo se estructura internamente un documento web, entendiendo la jerarquía de etiquetas, el funcionamiento de los atributos de clase y la diferencia entre la información semántica y el marcado de estilo. Este aprendizaje ha sido un objetivo fundamental del proyecto, convirtiendo la práctica no solo en un ejercicio de Flex, sino en una primera toma de contacto con HTML.

2. Desarrollo de la práctica

Esta práctica consiste en la automatización del flujo de datos desde su ubicación en los servidores de la Junta de Andalucía hasta su estructuración en memoria para ser consultados.

2.1. Obtención de la tabla de datos

Los datos epidemiológicos son dinámicos, ya que la Consejería de Salud y Consumo actualiza periódicamente las cifras de incidencia. Por ello, nuestro programa está diseñado para ser autónomo y robusto: al iniciarse, verifica la existencia del fichero local `data.html`. En caso de no encontrarlo, el sistema procede por sí mismo a la descarga de la versión más reciente en tiempo de ejecución, sin requerir intervención manual del usuario.

Para lograr esto, hemos integrado una llamada al sistema dentro del código C++ mediante la función `system()`. Esta ejecuta una orden que no solo descarga el archivo, sino que corrige su codificación al vuelo para adaptarla a Linux.

El comando implementado es el siguiente:

```
curl -s -L -k --data 'codConsulta=20072' \
https://www.juntadeandalucia.es/.../STPivot.jsp \
| iconv -f ISO-8859-1 -t UTF-8 > data.html
```

Es importante entender la configuración utilizada:

- **-L y -k:** Permiten seguir redirecciones del servidor y omitir la verificación estricta de certificados SSL (necesario para la conexión institucional).
- **-s (Silent):** Silencia la barra de progreso de la descarga para mantener la consola limpia.
- **Tubería (—) a iconv:** El flujo de datos descargado se pasa inmediatamente al comando `iconv`, que transforma los caracteres de la codificación original antigua (ISO-8859-1) a UTF-8 antes de guardarlos en el disco. Esto soluciona los problemas de visualización de tildes y caracteres especiales.

```
>No encuentro 'data.html'. Descargando...
>(El sistema descarga y convierte el archivo en
segundo plano)
```

Figura 1: Comportamiento del programa al detectar la ausencia de datos: descarga y conversión automática.

Al redirigir la salida final con `>data.html`, obtenemos un fichero local perfectamente legible y estructurado, que se convierte inmediatamente en la entrada (`yyin`) para nuestro analizador léxico generado por Flex.

2.2. Análisis del formato y discriminación de datos

Tras automatizar la descarga del fichero `data.html`, hemos tenido que analizar cómo es el código del mismo para poder procesarlo. Hemos notado que el servidor de la Junta de Andalucía (`STPivot.jsp`) no devuelve una tabla HTML visual estándar, sino un formato de exportación de datos estructurado basado en atributos.

Esto nos obligó a replantear nuestra estrategia de análisis léxico, identificando dos patrones clave que estructuran toda la información:

2.2.1. Identificación de Enfermedades (Etiquetas de Fila)

En lugar de utilizar filas de tabla convencionales (`<tr>`), el archivo define los encabezados de fila mediante el atributo `caption`. El patrón léxico que identifica el inicio de una posible enfermedad es:

```
<caption caption="Nombre de la Enfermedad" ... />
```

Un desafío crítico detectado en esta fase fue la ambigüedad semántica: el mismo patrón `caption` se utiliza tanto para las enfermedades (datos útiles) como para los metadatos de la tabla (“Sexo”, “Edad”, “Territorio”). Esto determinó la necesidad de implementar

```
<caption caption="Fiebre del Nilo Occidental" ... />
<caption caption="Sexo" ... />
```

Figura 2: Ejemplo del formato real encontrado en el fichero descargado.

un **filtro de exclusión** en la lógica del analizador para descartar estos "falsos positivos" y procesar únicamente las patologías reales.

2.2.2. Extracción de Valores Numéricos (Atributos de Valor)

Los datos estadísticos (número de casos) no se encuentran entre etiquetas de apertura y cierre, sino embebidos dentro de atributos `val`. El patrón identificado para la extracción de cifras es:

`val="154"`

Esta estructura simplifica el análisis léxico respecto a una tabla HTML tradicional, ya que elimina la necesidad de limpiar etiquetas de cierre o espacios en blanco circundantes. El analizador simplemente debe detectar el token `val="`, capturar la secuencia numérica subsiguiente y asignarla secuencialmente al vector correspondiente (Hombres, Mujeres o rangos de edad) según su orden de aparición.

2.2.3. Filtrado de Estructura

El resto del documento contiene etiquetas de definición XML/HTML y atributos de estilo que no aportan información epidemiológica. Nuestra estrategia consiste en ignorar cualquier token que no coincida explícitamente con los patrones de `caption` (filas) o `val` (datos), garantizando así una extracción limpia y eficiente.

3. Fundamentos Teóricos y Herramientas

Para dar solución al problema planteado, hemos aplicado los conceptos matemáticos y computacionales vistos en la asignatura como la generación automática de autómatas finitos y expresiones regulares.

3.1. El Generador Flex y los Autómatas Finitos

La herramienta central de este proyecto es Flex, usada para la extracción de información ya que traduce nuestras reglas léxicas en un Autómata Finito Determinista (AFD) optimizado.

Supone una ventaja frente a implementaciones manuales en C++ ya que, en lugar de tener un código con bucles anidados y condicionales con backtracking, nuestro código recorre

el fichero HTML una única vez. Esto hace que el procesamiento sea extremadamente rápido independientemente del tamaño del informe epidemiológico.

3.2. Expresiones Regulares y Lenguajes Formales

Para definir qué patrones debe detectar el autómata hemos hecho uso de las expresiones regulares. El mayor reto de la práctica ha sido trasladar la estructura visual y desordenada del HTML a esta estructura.

Para lograrlo, hemos hecho uso de los operadores fundamentales del álgebra de Kleene. Hemos utilizado la concatenación para unir las partes de las etiquetas HTML y la unión para definir alternativas léxicas. Y el más importante ha sido el uso de la Clausura de Kleene ya que gracias a este operador, pudimos definir patrones de exclusión que permiten al autómata ”saltar”sobre cualquier cantidad de atributos irrelevantes hasta encontrar el cierre de una etiqueta, permitiendo filtrar lo que no interesa del archivo fuente.

3.3. Integración con C++ y Librería STL

Mientras que Flex actúa como el reconocedor léxico, C++ actúa como el gestor semántico. Con esto podemos integrar el código generado por Flex (clase `yyFlexLexer`) con un entorno de programación.

Para el almacenamiento de los datos extraídos, hemos aplicado conceptos de estructuras de datos dinámicas utilizando STL. En lugar de utilizar arrays estáticos, hemos optado por el uso de vectores (`std::vector` para que así la memoria reservada crezca o decrezca en tiempo de ejecución según el volumen de datos de entrada, y no dependa el programa del tamaño de datos del fichero que se genera de la web `data.html`.

4. Diseño e Implementación de la Solución

4.1. Arquitectura del Sistema

Nuestra solución busca procesar los datos en tres etapas:

1. **Adquisición de Datos (Capa de Sistema):** El programa invoca al sistema operativo mediante la orden `curl` para realizar una petición HTTP al servidor de la Junta de Andalucía (`STPivot.jsp`). Esto nos permite trabajar con datos dinámicos y actualizados en lugar de recurrir a un fichero local estático.
2. **Análisis Léxico (Flex):** El fichero generado `data.html` es procesado por el escáner generado por `flex++` y se aplican las expresiones regulares para filtrar la entrada.
3. **Estructuración Semántica (C++):** A medida que Flex reconoce los símbolos de entrada, se ejecuta código C++ que valida la información y la organiza en estructuras de datos vectoriales en memoria dinámica.

4.2. Especificación de Patrones y Reglas Léxicas

Hemos definido un alfabeto de entrada y un conjunto de definiciones regulares que permiten al autómata finito determinista (AFD) generado por Flex identificar los lexemas relevantes.

4.2.1. Sección de Declaraciones y Estructuras de Datos

La primera parte del fichero `.l` establece el entorno de ejecución C++. Se incluyen las bibliotecas de flujos (`iostream`, `fstream`) y se definen los contenedores globales donde residirá la información.

```
/* --- Variables para guardar los datos --- */
/* Aquí vamos metiendo los nombres de las enfermedades que encontremos */
vector<string> Nombres;

/* Vectores para guardar los datos de cada columna (Hombres y Mujeres por edad) */
vector<string> H_Total, H_0_14, H_15_29, H_30_44, H_45_64, H_65_mas;
vector<string> M_Total, M_0_14, M_15_29, M_30_44, M_45_64, M_65_mas;

ifstream fichero;
int col = 0;   /* Para saber en qué columna de la tabla estamos */
int fila = 0;  /* Contador de filas */
bool guardar = false; /* Interruptor para saber si guardamos el dato o pasamos */
```

Listing 1: Declaración de vectores paralelos para el almacenamiento de datos

Hemos optado por un diseño de vectores. Se define un vector principal `Nombres` y 12 auxiliares (`H_Total`, `H_0_14`, etc.). Esta estructura simula una matriz bidimensional donde el índice i vincula el nombre de la patología con sus estadísticas correspondientes.

4.2.2. Definiciones Regulares

En la sección de definiciones de Flex, hemos construido patrones que facilitan la legibilidad y el mantenimiento de la gramática léxica.

```
/* --- Definiciones básicas para Flex --- */
digito      [0-9]
mayus       [A-Z]
letra       [a-z]
cualquiera  .

/* Patrones para pillar nombres (palabras sueltas o compuestas) */
palabra     (({mayus}+|{cualquiera}){letra}*{cualquiera}*{letra}*)
compuesta   ({palabra}\b{palabra})
texto_enf   ({palabra}|{compuesta}+)

/* Etiquetas del HTML que nos interesan */
TAG_CAP     ("<caption caption=")
TAG_FIN     ("/>")
TAG_VAL     ("val=")
```

```
COMILLA      ("\"")
```

Listing 2: Definiciones regulares y patrones de atributos

- **Codificación (`cualquiera`):** Uno de los retos encontrados fue la codificación de caracteres especiales en la respuesta del servidor. Las tildes y eñes a menudo aparecen con codificaciones no estándar. Para solucionar esto a nivel de autómata, definimos la regla `cualquiera .`. El punto (`.`) actúa como un comodín léxico. Esto permite que los patrones reconozcan palabras como Infección o Sarampión independientemente de la respuesta del servidor.
- **Identificadores (`texto_enf`):** Definimos patrones recursivos (`palabra`, `compuesta`) para capturar nombres de enfermedades complejos. Esta expresión regular acepta cadenas que comiencen por mayúscula o un carácter especial, seguidas de secuencias alfabéticas mixtas.
- **Marcado (`TAG_CAP` y `TAG_VAL`):** A diferencia de un parser HTML visual, analizamos la estructura interna de los datos exportados. Identificamos dos delimitadores: `TAG_CAP` (inicio de fila) y `TAG_VAL` (dato de celda).

4.2.3. Sección de Reglas y Autómata Implícito

Regla 1: Reconocimiento y Filtrado Cuando el escáner encuentra el patrón `TAG_CAP`, extrae el contenido entre comillas. Sin embargo, el fichero crudo contiene ruido semántico (cabeceras como Sexo, Edad, Territorio). Implementamos un filtro en C++ dentro de la acción semántica:

```
{TAG_CAP}{COMILLA}{texto_enf}{COMILLA}{TAG_FIN} {
    /* Regla 1: Hemos encontrado un nombre de enfermedad */
    if(fila >= 0){
        string t = limpiar(string(YYText()));

        /* Filtro manual para quitar cabeceras y texto que no es
   enfermedad */
        if(t.find("Sexo") == string::npos && t.find("Hombres") ==
   string::npos &&
           t.find("Mujeres") == string::npos && t.find("Edad") ==
   string::npos &&
           t.find("TOTAL") == string::npos && t.find("Total") == string
   ::npos &&
           t.find("Andaluc") == string::npos && t.find("Unidad") ==
   string::npos &&
           t.find("Fuente") == string::npos && t.find("Consej") ==
   string::npos &&
           t.find("A=") == string::npos && t.find("Nivel") == string::
   npos &&
           t.find("Territorio") == string::npos && t.find("Medida") ==
   string::npos &&
           t.find("Casos") == string::npos && t.find("EDO") == string::
   npos &&
           t.find("os") == string::npos && t.length() > 3) {

            // Si pasa el filtro, activamos el guardado
```

```
            guardar = true;
            Nombres.push_back(t);
        } else {
            guardar = false;
        }
    }
    fila++;

}
```

Listing 3: Regla de reconocimiento de enfermedad con filtro de exclusión

Solo si la cadena supera este filtro, se activa la bandera de estado `guardar = true`. Esto previene que el autómata procese números pertenecientes a filas de cabecera, asegurando la integridad de los datos estadísticos.

Regla 2: Autómata de Distribución de Datos Para la lectura de los valores numéricos (`TAG_VAL`) debemos tener en cuenta que el fichero es un flujo lineal de datos, pero nuestra estructura es de tabla (12 columnas por fila). Para resolver esto, hemos implementado un **autómata finito determinista implícito** mediante la variable de estado `col`:

```
{TAG_VAL}{COMILLA}[0-9.]+{COMILLA} {
    /* Regla 2: Hemos encontrado un valor numérico */
    col++;
    string v = limpiar(string(YYText()));

    // Solo guardamos si estamos en una fila válida (enfermedad real)
    if(guardar){
        switch (col){
            /* Hombres */
            case 1: H_Total.push_back(v); break;
            case 2: H_0_14.push_back(v); break;
            case 3: H_15_29.push_back(v); break;
            case 4: H_30_44.push_back(v); break;
            case 5: H_45_64.push_back(v); break;
            case 6: H_65_mas.push_back(v); break;
            /* Mujeres */
            case 7: M_Total.push_back(v); break;
            case 8: M_0_14.push_back(v); break;
            case 9: M_15_29.push_back(v); break;
            case 10: M_30_44.push_back(v); break;
            case 11: M_45_64.push_back(v); break;
            case 12:
                M_65_mas.push_back(v);
                col = 0; // Reiniciamos columna
                break;
        }
    } else {
        if (col >= 12) col = 0;
    }
}
```

Listing 4: Lógica de distribución de columnas mediante Switch-Case

Cada vez que se reconoce un token `TAG_VAL` válido:

1. Se incrementa el contador de estado (`col`).
2. Se ejecuta una estructura de selección múltiple (`switch`) que actúa como función de transición:
   - Estado 1 → Almacenar en `H_Total`
   - Estado 2 → Almacenar en `H_0_14`
   - ...
   - Estado 12 → Almacenar en `M_65_mas` y Transición a Estado Inicial (`col = 0`).

Esta lógica permite reconstruir la tabla bidimensional original a partir de un flujo unidimensional de símbolos.

4.3. Interfaz y Gestión de Archivos

La función principal (`main`) implementa una lógica de recuperación ante fallos: intenta abrir el flujo de entrada `data.html`. En caso de fallo (fichero inexistente), el programa cambia dinámicamente su comportamiento, ejecutando la llamada al sistema (`system()`) para descargar los datos mediante `curl` con las banderas `-L` (seguir redirecciones) y `-k` (ignorar SSL), garantizando la obtención del recurso.

Finalmente, se presenta una interfaz de texto (CLI) que permite la navegación indexada por los vectores cargados, formateando la salida mediante la biblioteca `iomanip` para una visualización clara de las tablas epidemiológicas.

5. Implementación Técnica y Compilación

Para facilitar la compilación del proyecto y no tener que escribir los comandos largos cada vez, hemos creado un archivo `Makefile`. Esto nos permite compilar tanto el código que genera Flex como nuestro programa en C++ simplemente escribiendo `make` en la terminal.

5.1. El archivo Makefile

Nuestro `Makefile` está diseñado de forma modular utilizando variables para facilitar cambios futuros (como cambiar el compilador o el nombre del ejecutable).

Las reglas definidas son:

- **Variables:** Definimos `TARGET` (nombre del ejecutable), `CXX` (compilador C++) y `FLEX` (generador léxico).
- **all:** Es la regla por defecto. Asegura que el objetivo final (`ANALIZADOR`) esté construido.
- **lex.yy.cc:** Genera el código fuente C++ a partir de las expresiones regulares usando `flex++`.

- **clean:** Elimina los archivos generados y el fichero `data.html` para reiniciar las pruebas de descarga.
- **run:** Compila si es necesario y ejecuta el programa inmediatamente.

```
TARGET = ANALIZADOR
CXX = g++
FLEX = flex++

all: $(TARGET)

$(TARGET): lex.yy.cc
    $(CXX) lex.yy.cc -o $(TARGET)

lex.yy.cc: analizador.l
    $(FLEX) analizador.l

clean:
    rm -f $(TARGET) lex.yy.cc data.html

run: $(TARGET)
    ./$(TARGET)
```

Listing 5: Código del archivo Makefile

5.2. Código Fuente: analizador.l

El código fuente `analizador.l` implementa un sistema híbrido que une la potencia de Flex para el reconocimiento de patrones con la gestión de memoria de C++. Su lógica se basa en dos reglas principales: una para detectar los nombres de las enfermedades (buscando la etiqueta `caption`) y otra para capturar las cifras (etiqueta `val`). Al encontrar estos patrones, el programa 'limpia' el ruido HTML y distribuye los datos en vectores dinámicos. Además, destaca la función `main`, que incluye una llamada al sistema para descargar y convertir automáticamente la codificación del archivo (usando `iconv`), solucionando así el problema de compatibilidad de caracteres entre el servidor de la Junta y la terminal Linux.

5.3. Proceso de Compilación

Gracias al archivo Makefile, la compilación es muy sencilla. Como se ve en la siguiente captura, al ejecutar `make`, el sistema traduce primero el archivo `.l` y luego lo compila con el código de Flex.

```
usuario@usuario-vb:~$ make
flex++ analizador.l
g++ lex.yy.cc -o ANALIZADOR
```

Figura 3: Ejecución del comando make en la terminal de Ubuntu.

6. Pruebas y Resultados

En esta sección mostramos las pruebas que hemos hecho para comprobar que el programa funciona bien y maneja correctamente la descarga de datos y los caracteres especiales.

6.1. Integración de CURL y Solución de Codificación

Uno de los problemas que nos encontramos fue que los datos venían de una web segura (HTTPS) y que, al descargarlos, las tildes salían mal porque el servidor usa una codificación antigua (ISO-8859-1) y Linux usa moderna (UTF-8).

Para solucionarlo, aprendimos a usar el comando `curl` con una tubería (pipe) hacia `iconv`.

- **curl -L -k**: Para seguir redirecciones y saltar la validación SSL estricta.
- **iconv -f ISO-8859-1 -t UTF-8**: Para convertir el archivo al vuelo y que las tildes se vean bien.

En la Figura 4 se ve cómo el programa detecta que falta el archivo y lo descarga automáticamente.

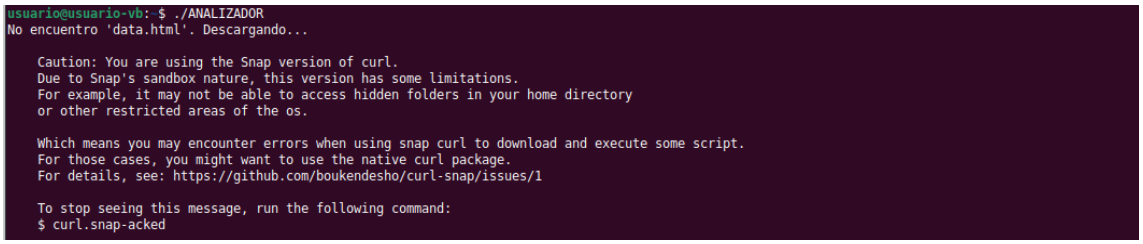

Figura 4: El programa descarga los datos automáticamente usando CURL e iconv.

6.2. Menú Principal y Listado

Una vez cargados los datos, el programa muestra el menú principal limpio. En la Figura 5 se ve la interfaz del usuario.

Figura 5: Menú principal del Sistema de Vigilancia EDO.

Si seleccionamos la opción 1, obtenemos el listado de enfermedades. Como se observa en la Figura 6, las tildes y eñes se muestran correctamente gracias a la conversión realizada durante la descarga.

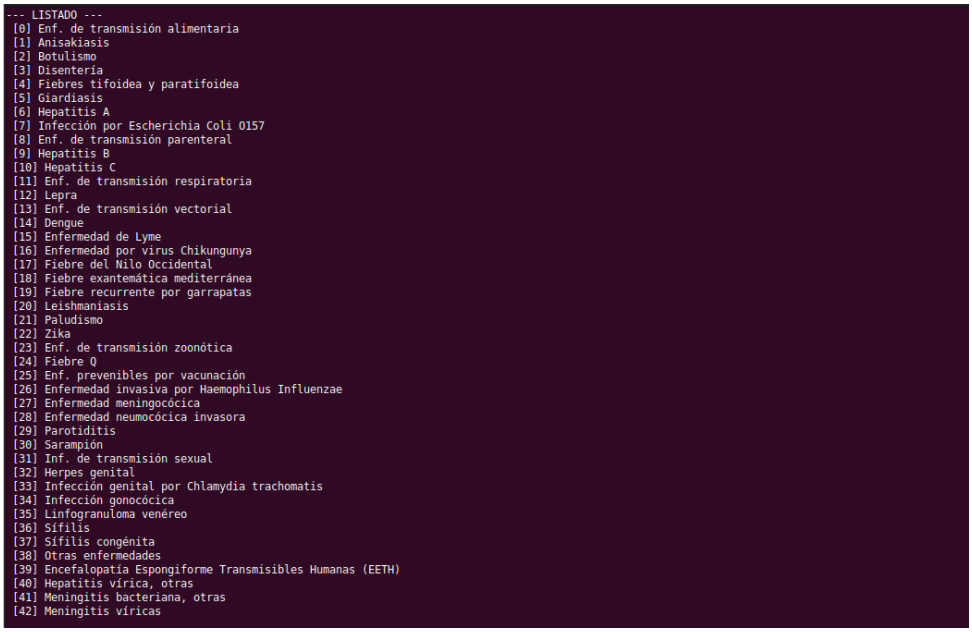

Figura 6: Listado de enfermedades detectadas. Se observan caracteres como 'Anisakiasis' correctamente formateados.

6.3. Validación de Datos Detallados

Por último, comprobamos que los datos numéricos se han guardado bien. En la Figura 7 pedimos los detalles de la Anisakiasis (ID 1). El programa muestra la tabla perfectamente alineada, separando Hombres y Mujeres por rangos de edad.

Figura 7: Ficha técnica detallada de una enfermedad mostrando los datos demográficos.

6.4. Reinicio del Sistema y Finalización

Para facilitar las pruebas y verificar la robustez del mecanismo de descarga, implementamos la Opción 3 (Borrar datos y salir).

Esta funcionalidad es crítica para el ciclo de desarrollo, ya que elimina el fichero local `data.html` mediante una llamada al sistema. Como se observa en la Figura 8, al seleccionar esta opción, el programa confirma la eliminación y se cierra. Esto nos permite volver a ejecutar el programa y comprobar que el sistema de recuperación con `curl` sigue funcionando correctamente desde cero.

Figura 8: Ejecución de la opción de borrado para reiniciar el entorno de pruebas.

Finalmente, la Opción 4 permite terminar la ejecución del bucle principal de manera ordenada, devolviendo el control al sistema operativo sin errores.

7. Conclusión

En conclusión, la realización de esta práctica nos ha permitido comprender la utilidad real de herramientas como Flex para el procesamiento de ficheros de texto y la extracción de información.

Aunque al principio nos encontramos con dificultades debido al formato del archivo HTML de la Junta de Andalucía (que contenía muchas etiquetas y palabras ”basura”que ensuciaban la lectura), logramos solucionar el problema ajustando las expresiones regulares y añadiendo filtros en el código C++.

Gracias a esto, hemos conseguido automatizar la obtención de los datos de las Enfermedades de Declaración Obligatoria (EDO) por edad y sexo, algo que sería muy tedioso de hacer a mano. En definitiva, este trabajo nos ha servido para aprender a integrar un analizador léxico con estructuras de datos en C++ y ver cómo se aplican los modelos de computación en problemas reales.

Referencias

[1] The Flex Project. *Flex: The Fast Lexical Analyzer Manual*. Disponible en: https://westes.github.io/flex/manual/

[2] Junta de Andalucía. Consejería de Salud y Consumo. *Banco de Datos Estadísticos de Andalucía (BADEA): Enfermedades de Declaración Obligatoria (EDO)*. Disponible en: https://www.juntadeandalucia.es/institutodeestadisticaycartografia/badea/

[3] Stenberg, D. et al. *curl: Command line tool and library for transferring data with URLs*. Disponible en: https://curl.se/docs/manpage.html

[4] CppReference. *C++ Standard Library documentation: std::vector, std::fstream*. Disponible en: https://en.cppreference.com/

[5] Free Software Foundation. *GNU libiconv: Character set conversion library*. Disponible en: https://www.gnu.org/software/libiconv/