UNIVERSIDAD DE GRANADA

**Práctica 2. Modelos de Computación**

**Ingeniería Informática (Grupo A, curso 21/22)**

**Implementación de escáneres mediante expresiones regular**

# 1. Introducción

Normalmente, `flex` se usa en la primera etapa necesaria a la hora de elaborar un compilador, un intérprete, un emulador o cualquier otro tipo de herramienta que necesite procesar un fichero de entrada para poder cumplir su misión. Otra utilidad de `flex` consiste en ser una herramienta que nos permite ejecutar acciones tras la localización de cadenas de entrada que emparejan con expresiones regulares. En esta práctica nos centraremos en esta segunda utilidad de `flex`. Se le pedirá al estudiante la realización de un trabajo práctico de procesamiento de un fichero que involucre el uso de `flex` para localizar ciertas cadenas en el fichero y ejecutar una acción correspondiente con cada una de ellas. Por ejemplo, localizar y borrar direcciones de correo electrónico en una página web, cambiar de color ciertas palabras en un fichero html , etc. El alumno deberá plantear su propio trabajo práctico de procesamiento de ficheros usando `flex`. A continuación se describirá una breve introducción sobre el `flex` y sus conceptos asociados que podrá servir de ayuda al estudiante para la realización de su práctica.

# 2. Introducción a flex

`flex` es una herramienta de los sistemas Linux que nos va a permitir generar un escáner en código C que luego podremos compilar y enlazar con nuestro programa. En esta práctica utilizaremos `flex` con la opción para generar código C++ puesto que es el lenguaje estudiado en las asignaturas de programación. Esto se consigue simplemente realizando la llamada al programa con `flex++`.

La principal característica de `flex` es que nos va a permitir asociar acciones descritas en C ó C++, a la localización de las Expresiones Regulares que le hayamos definido. `flex` se apoya en una plantilla que recibe como parámetro y que deberemos diseñar con cuidado. En esta plantilla le indicaremos las expresiones regulares que debe localizar y las acciones asociadas a cada una de ellas. A partir de esta plantilla, `flex` genera código fuente en C ó C++. Este código contiene una función llamada yylex(), que localiza cadenas en la entrada que se ajustan a las expresiones regulares definidas, realizando entonces las acciones asociadas a dicho patrón. Un manual on-line se puede encontrar en `https://westes.github.io/flex/manual/`

# 3. Expresiones regulares en Linux

Las expresiones regulares nos permiten hacer búsquedas contextuales y modificaciones sobre textos. Una expresión regular en Linux es un patrón que describe un conjunto de cadenas de caracteres. Por ejemplo, el patrón `aba*.txt` describe el conjunto de cadenas de caracteres que comienzan con `aba`, contienen cualquier otro grupo de caracteres, luego un punto, y finalmente la cadena `txt`.

Una Expresión Regular nos sirve para definir lenguajes, imponiendo restricciones sobre las secuencias de caracteres que se permiten en este lenguaje. Por tanto una Expresión Regular estará formada por el conjunto de caracteres del alfabeto original, más un pequeño conjunto de caracteres extra (meta-caracteres) que nos permitirán definir estas restricciones.

El conjunto de metacaracteres para expresiones regulares es el siguiente:

```
\^$.[]{}|()*+?
```

Estos caracteres, en una expresión regular, son interpretados de una manera especial y no como los caracteres que normalmente representan. Una búsqueda que implique alguno de estos caracteres obligará a usar el carácter `\`. Por ejemplo, en una expresión regular, el carácter `.` representa "un carácter cualquiera". Pero si escribimos `\.`, estamos representando el carácter `.` tal cual, sin significado adicional.

| Expresión Regular | Significado |
|---|---|
| Caracteres normales | Ellos mismos |
| `.` | Un carácter cualquiera excepto nueva línea |
| `r*` | `r` debe aparecer cero o más veces |
| `r+` | `r` debe aparecer una o más veces |
| `r?` | `r` debe aparecer cero o una vez |
| `r1\|r2` | La expresión regular `r1` o la `r2` |
| `^` | Ubicado al principio de la línea |
| `$` | Ubicado al final de la línea |
| `-` | Dentro de un conjunto de caracteres escrito entre corchetes, podemos especificar un rango (ej. `[a-zA-Z0-9]`). |
| `[...]` | Un carácter cualquiera de los caracteres en `...` Acepta intervalos del tipo `a-z`, `0-9`, `A-Z`, etc. |
| `[^...]` | Un carácter distinto de los caracteres en `...` Acepta intervalos del tipo `a-z`, `0-9`, `A-Z`, etc. |
| `(...)` | Agrupación de los elementos dentro del paréntesis |
| `\n` | Carácter de salto de línea |
| `\t` | Carácter de tabulación |
| `r1r2` | La expresión regular `r1` seguida de la expresión regular `r2` |
| `r{n}` | $n$ ocurrencias de la expresión regular `r` |
| `r{n,}` | $n$ o más ocurrencias de la expresión regular `r` |
| `r{,m}` | Cero o a lo sumo $m$ ocurrencias de la expresión regular `r` |
| `r{n,m}` | $n$ o más ocurrencias de la expresión regular `r`, pero a lo sumo $m$ |
| `{nombre}` | Se sustituye la expresión regular llamada `nombre` |
| `"..."` | Los caracteres entre comillas literalmente |

Algunos ejemplos de expresiones regulares

| Expresión Regular | Significado |
|---|---|
| `a.b` | `axb aab abb aSb a#b` ... |
| `a..b` | `axxb aaab abbb a4$b` ... |
| `[abc]` | `a b c` (cadenas de un carácter) |
| `[aA]` | `a A` (cadenas de un carácter) |
| `[aA][bB]` | `ab Ab aB AB` (cadenas de dos caracteres) |
| `[0123456789]` | `0 1 2 3 4 5 6 7 8 9` |
| `[0-9]` | `0 1 2 3 4 5 6 7 8 9` |
| `[A-Za-z]` | `A B C ... Z a b c ... z` |
| `[0-9][0-9][0-9]` | `000 001 ... 009 010 ... 019 100 .. 999` |
| `[0-9]*` | Cadena vacía `0 1 9 00 99 123 456 999 9999` ... |
| `[0-9][0-9]*` | `0 1 9 00 99 123 456 999 9999 99999 99999999` ... |
| `^1` | Cadenas que empiecen por el símbolo `1` |
| `^[12]` | Cadenas que empiezan por `1` o por `2` |
| `^[^12]` | Todas las cadenas menos las que empiezan por `1` o por `2` |
| `(123\|124)$` | Cadenas que acaban con `123` o con `124` |
| `[0-9]+` | `0 1 9 00 99 123 456 999 9999 99999 99999999` ... |
| `[0-9]?` | Cadena vacía `0 1 2 ... 9` |
| `(ab)*` | Cadena vacía `ab abab ababab` ... |
| `^[0-9]?b` | `b 0b 1b 2b ... 9b` |
| `([0-9]+ab)*` | Cadena vacía `1234ab 9ab9ab9ab 9876543210ab 99ab99ab` ... |

# 4. Estructura de un fichero flex

La plantilla en la que `flex` se va a apoyar para generar el código C++, y donde nosotros deberemos describir toda la funcionalidad requerida, va a ser un fichero de texto plano con una estructura bien definida, donde iremos describiendo las expresiones regulares y las acciones asociadas a ella. La estructura de la plantilla es la siguiente:

```
Declaraciones

%%

Reglas

%%

Procedimientos de Usuario
```

Se compone de tres secciones con estructuras distintas y claramente delimitadas por una línea en la que lo único que aparece es la cadena `%%`.

Las secciones de `Declaraciones` y la de `Procedimientos de Usuario` son opcionales, mientras que la de `Reglas` es obligatoria (aunque se encuentre vacía), con lo que tenemos que la plantilla más pequeña que podemos definir es:

```
%%
```

Esta plantilla, introducida en `flex`, generaría un programa C++ donde el contenido de la entrada estándar sería copiado en la salida estándar por la aplicación de las reglas y acciones por defecto. `flex` va a actuar como un pre-procesador que va a trasformar las definiciones de esta plantilla en un fichero de código C++.

## 4.1. La sección de Declaraciones

En la sección de `Declaraciones` podemos encontrar dos tipos de declaraciones bien diferenciados:

- Un bloque donde le indicaremos al pre-procesador que lo que estamos definiendo queremos que aparezca "tal cual" en el fichero C++ generado. Es un bloque de copia delimitado por las secuencias `%{` y `%}` donde podemos indicar la inclusión de los ficheros de cabecera necesarios, o la declaración de variables globales, o declaraciones procedimientos descritos en la sección de `Procedimientos de Usuario`. Por ejemplo:

  ```
  %{
  #include <iostream>
  #include <fstream>

  using namespace std;

  ifstream fichero;
  int  nc, np, nl;
  void escribir_datos (int dato1, int dato2, int dato3);
  %}
  ```

- Un bloque de definición de "alias", donde "pondremos nombre" a algunas de las expresiones regulares utilizadas. En este bloque, aparecerá AL COMIENZO DE LA LÍNEA el nombre con el que bautizaremos a esa expresión regular y SEPARADO POR UN TABULADOR (al menos), indicaremos la definición de la expresión regular. Para utilizar dichos nombres en vez de las expresiones regulares debemos escribirlos entre llaves. Por ejemplo:

```
linea       \n
entero      \t
DIGITO      [0-9]
ID          [a-z][a-z0-9]*
DECIMAL     {DIGITO}+"."{DIGITO}*
```

Estos bloques pueden aparecer en cualquier orden, y pueden aparecer varios de ellos a lo largo de la sección de declaraciones. Recordemos que esta sección puede aparecer vacía.

## 4.2. La sección de Reglas

En la sección de `Reglas` sólo permitiremos un único tipo de escritura. Las reglas se definen como sigue:

```
Expresión_Regular   {acciones escritas en C++}
```

AL COMIENZO DE LA LÍNEA se indica la expresión regular, seguida inmediatamente por uno o varios TABULADORES, hasta llegar al conjunto de acciones en C++ que deben ir encerrados en un bloque de llaves.

A la hora de escribir las expresiones regulares podemos hacer uso de los acrónimos dados en la sección de `Declaraciones`, escribiéndolos entre llaves, y mezclándolos con la sintaxis general de las expresiones regulares. Por ejemplo, `^{linea}`.

Si las acciones descritas queremos que aparezcan en varias líneas debido a su tamaño, debemos comenzar cada una de esas líneas con al menos un carácter de tabulación. Si queremos incorporar algún comentario en C++ en una o varias líneas debemos comenzar cada una de esas líneas con al menos un carácter de tabulación. Un ejemplo del contenido de la sección de `Reglas` podría ser:

```
[^ \t\n]+  { np++; nc += yyleng; }
[ \t]+    { nc += yyleng; }
\n     { nl++; nc++; }
```

Como normas para la identificación de expresiones regulares, `flex` sigue las siguientes:

- Siempre intenta encajar una expresión regular con la cadena más larga posible,
- En caso de conflicto entre expresiones regulares (pueden aplicarse dos o más para una misma cadena de entrada), `flex` se guía por estricto orden de declaración de las reglas.

Existe una regla por defecto, que es:

```
.   {ECHO;}
```

Esta regla se aplica en el caso de que la entrada no encaje con ninguna de las reglas. Lo que hace es imprimir en la salida estándar el carácter que no encaja con ninguna regla. Si queremos modificar este comportamiento tan solo debemos sobrescribir la regla `.` con la acción deseada (`{}` si no queremos que haga nada).

## 4.3. La sección de Procedimientos de Usuario

En la sección de `Procedimientos de Usuario` escribiremos en C++ sin ninguna restricción aquellos procedimientos que hayamos necesitado en la sección de Reglas. Todo lo que aparezca en esta sección será incorporado "tal cual" al final del fichero `lex.yy.cc`

No debemos olvidar como concepto de C++, que si la implementación de los procedimientos se realiza "después" de su invocación (en el caso de `flex`, lo más probable es que se hayan invocado en la sección de reglas), debemos haberlos declarado previamente. Para ello no debemos olvidar declararlos en la sección de `Declaraciones`. Por ejemplo, en nuestro caso:

```
void escribir_datos (int dato1, int dato2, int dato3);
```

Como función típica a ser descrita en una plantilla `flex`, aparece el método principal (main). Un ejemplo de método "main" típico es aquel que acepta un nombre de fichero como fichero de entrada. Esto es:

```
int main (int argc, char *argv[])
{
  if (argc == 2)
   {
     fichero.open (argv[1]);
     if (fichero==0)
     {
       cout << "error de lectura" << endl;
       exit (1);
     }
    }
   else exit(1);

   nc = np = nl = 0;

yyFlexLexer flujo (&fichero,0);
flujo.yylex();
escribir_datos(nc,np,nl);

  return 0;
}

void escribir_datos (int dato1, int dato2, int dato3)
{
    cout << "Num_lineas = " << dato3 << endl;
    cout << "Num_palabras = " << dato2 << endl;
cout << "Num_caracteres = " << dato1 << endl;
}
```

# 5. Clases, variables, funciones, procedimientos y macros de flex

El analizador generado puede utilizar ciertas variables y funciones miembro de dos clases de C++. Estas se declaran en el fichero de cabecera `FlexLexer.h`, que se incluye por defecto al generar el analizador.

La primera clase, `FlexLexer`, provee las siguientes funciones miembro:

- `const char* YYText()` retorna el texto del token reconocido más recientemente. También se puede utilizar la variable de tipo `char *`, `yytext`, que realiza la misma función cuando se utiliza el lenguaje C.
- `int YYLeng()` retorna la longitud del token reconocido más recientemente. También se puede utilizar la variable de tipo `int`, `yyleng`, que realiza la misma función cuando se utiliza el lenguaje C.
- `int lineno() const` retorna el número de línea de entrada actual.

La segunda clase definida, `yyFlexLexer`, se deriva de `FlexLexer`. Esta define las siguientes funciones miembro adicionales:

- `yyFlexLexer( istream* arg_yyin = 0, ostream* arg_yyout = 0 )` construye un objeto `yyFlexLexer` usando los flujos dados para la entrada y salida. Si no se especifica, los flujos se establecen por defecto a `cin` y `cout`, respectivamente.
- `virtual int yylex()` analiza el flujo de entrada, consumiendo tokens, hasta que la acción de una regla retorne un valor. En otras palabras, inicia el analizador.
- `yywrap ()` Se invoca automáticamente al encontrar el final del fichero de entrada. Cuando el analizador reciba una indicación de fin-de-fichero, entonces esta comprueba la función `yywrap()`. Si `yywrap()` devuelve falso (cero), entonces se asume que la función ha ido más allá y ha preparado la entrada de otro fichero y

el análisis continúa. Si este retorna verdadero (no-cero), entonces el analizador termina. Si sólo se piensa utilizar un fichero de entrada, se puede evitar su llamada con la opción `noyywrap`. Esto es, añadiendo la línea `%option noyywrap`.

# 6. Ejemplo

El siguiente ejemplo de plantilla nos permite contar los caracteres, palabras y líneas de un fichero:

```
%option noyywrap

             /*----- Seccion de Declaraciones ----------------*/

%{
#include <iostream>
#include <fstream>

using namespace std;

ifstream fichero;
int  nc, np, nl;
void escribir_datos (int dato1, int dato2, int dato3);
%}

%%
                 /*----- Seccion de Reglas ----------------*/

[^ \t\n]+  { np++; nc += yyleng; }
[ \t]+    { nc += yyleng; }
\n     { nl++; nc++; }


%%
                /*----- Seccion de Procedimientos --------*/

int main (int argc, char *argv[])
{
  if (argc == 2)
   {
     fichero.open (argv[1]);
     if (fichero==0)
     {
       cout << "error de lecura" << endl;
       exit (1);
     }
    }
   else exit(1);

   nc = np = nl = 0;

yyFlexLexer flujo (&fichero,0);
flujo.yylex();
escribir_datos(nc,np,nl);

  return 0;
```

```
}

void escribir_datos (int dato1, int dato2, int dato3)
{
    cout << "Num_lineas = " << dato3 << endl;
cout << "Num_palabras = " << dato2 << endl;
cout << "Num_caracteres = " << dato1 << endl;
}
```

# 7. El proceso de compilación

Los pasos para obtener un fichero ejecutable usando `flex` son los siguientes:

1. Crear la plantilla `flex`. Por ejemplo, `plantilla.l`.
2. Invocar la herramienta `flex` mediante el comando shell de Linux:

   ```
   flex++ plantilla.l
   ```

   Este paso nos genera un fichero C++, denominado `lex.yy.cc` que contiene todo el código necesario para compilar nuestra aplicación.
3. Compilar el archivo `lex.yy.cc` junto a las librerías adecuadas, por ejemplo,

   ```
   g++ lex.yy.cc -o prog
   ```
4. Ya sólo nos queda ejecutar el ejecutable `prog` sobre un fichero de entrada.

   ```
   ./prog <Nombre_Fichero>
   ```

# 8. Tareas a realizar

1. Formar un grupo de trabajo compuesto por una, dos o tres personas.
2. Cada grupo de trabajo debe pensar un problema original de procesamiento de textos. Para la resolución de este problema debe ser apropiado el uso de `flex`, o sea, se debe resolver mediante el emparejamiento de cadenas con expresiones regulares y la asociación de acciones a cada emparejamiento.
3. Cada grupo debe resolver el problema propuesto usando `flex`. Se deberá realizar una memoria donde se presente una descripción del problema y su solución, además de entregar electrónicamente los ficheros de texto con la implementación de la solución.
4. Esta práctica deberá ser entregada antes del día 31 de Diciembre de 2021. Se entregará a través de la plataforma PRADO en un fichero `.zip` conteniendo todos los archivos de esta práctica. Sólo es necesario que lo entregue uno de los componentes del grupo (aunque en la memoria debe especificarse quiénes son los miembros del grupo).