

# Trabajo de Laboratorio de CL-1

## 1. El Flex

El Flex es un programa que traduce una especificación léxica (el *fuentes* Flex, que, básicamente, contiene expresiones regulares con acciones C asociadas) en un programa C que realiza el análisis léxico. A continuación damos el formato de un fichero de entrada de Flex, junto con el C generado:

Entrada Flex:		Fichero C generado
=====		=====
%{		
Declaraciones C	----->	Declaraciones C /* se copian tal cual */
%}		
Definiciones	\	int yylex() {
%option noyywrap	\	// Función que lee la entrada, y cada vez que
%%	-->	// encuentra un prefijo que está en el lenguaje de
Reglas del tipo:	/	// una ExpReg, ejecuta el CódigoC correspondiente.
ExpReg {CodigoC}	/	}
%%		
Rutinas C	----->	Rutinas C /* se copian tal cual */

En las **Definiciones** podemos escribir cosas como:

```
DIGITO    [0-9]
LETRA     [A-Za-z]
```

asociando a un nombre (por ejemplo, **DIGITO**) una expresión regular (en el ejemplo, **[0-9]**, que equivale a cualquier carácter ASCII en el rango dado).

En el apartado de **Reglas**, escribiremos expresiones regulares con acciones C asociadas. En las expresiones regulares podemos usar *entre llaves* (es decir, entre { y } ) los nombres definidos en el apartado de **Definiciones**. Por ejemplo:

```
\n                { numlinea++; }
{LETRA}({LETRA}|{DIGITO})* { tratarid(yytext,yylen); }
```

En este ejemplo incrementamos un contador de números de línea cada vez que llega un salto de línea (este contador puede servirnos después para poder decir en qué línea se produce un error). En la segunda línea del ejemplo, cada vez que se detecta un identificador lo tratamos con el procedimiento nuestro **tratarid**. Este procedimiento utiliza dos variables importantes de Flex: **yytext** es de tipo **char \*yytext** y contiene la cadena reconocida en esta expresión regular (en este caso, el identificador); **yylen** es de tipo **int** y contiene la longitud de la cadena en **yytext**.

El código C creado por el Flex contiene una función C llamada **yylex()**, que busca el prefijo de la entrada más largo que corresponde a alguna expresión regular del apartado de reglas, y ejecutará la acción asociada a la primera expresión regular que reconoce este prefijo. Después de ejecutar la acción, el **yylex()** continuará con la parte restante de la entrada (excepto si la acción contiene un **return**, como se verá en la siguiente hoja de laboratorio). Si no hay ningún prefijo reconocido como alguna expresión regular, entonces se ignora el primer carácter de la entrada (se escribe por **yyout**, que por defecto es la salida estándar) y se vuelve a intentar.

En el apartado de rutinas C tendremos funciones C, como **tratarid**. Si queremos utilizar la función **yylex()** independientemente (y no, como es también habitual, desde el analizador sintáctico), también incluiremos en este apartado un **main** que llame a la función **yylex()**.

### Cómo especificar las expresiones regulares en Flex?

Las sintaxis es similar a la habitual para expresiones regulares: **ab\*** denota  $a.(b^*)$ , **aa|b** denota  $(a.a) \cup b$ , **a|b\*** denota  $a \cup (b^*)$ , **a+** denota  $a(a)^*$ . La prioridad de las operaciones va desde **\*** (máxima) hasta **|** (mínima), por lo que hay que introducir paréntesis para algunos casos: **(ab)\*** denota  $(ab)^*$ . También podemos expresar pertenencias a conjuntos de caracteres e intervalos: **[abc]** denota  $a|b|c$ , **[a-zA-Z]** denota  $[a-z] \cup [A-Z]$ . Finalmente, hay meta-caracteres: **[^ab]** denota cualquier carácter excepto **a** y **b**, **\n** es un salto de línea, **\t** es un tabulador, el punto

. denota cualquier carácter excepto el salto de línea (es decir, `[^\n]`) y el `\` quita el significado especial a cualquier meta-carácter que se escriba a continuación; por ejemplo, para denotar el carácter `^`, escribiremos `\^`. Asimismo, el carácter blanco lo podemos describir con un `\` seguido de un espacio, o con `[ ]`.

## Cómo compilar un fuente Flex en Linux?

Supongamos que el fuente Flex se llama *nombre.1*. El comando

```
flex -t -8 nombre.1 > nombre.c
```

(con `-t` para poder redireccionar la salida y `-8` para poder usar caracteres ASCII con acentos) genera un fichero *nombre.c*. También existe la opción `-i`, que lo hace insensible a mayúsculas/minúsculas en las expresiones regulares; por ejemplo la expresión regular `else` admitirá tanto `ELSE` como `Else` como `else`, etc. El fichero *nombre.c* se puede compilar, como cualquier otro programa C, con `gcc nombre.c -o nombre`. Esto creará un ejecutable *nombre*, que leerá los caracteres de entrada de *yyin*, que por defecto es `stdin` (el teclado). Si queremos que lea de un fichero llamado *entrada*, podemos incluir en el `main`, antes de llamar a `yylex()`, la instrucción `yyin = fopen("entrada", "r");`.

## Un ejemplo

En este ejemplo, la primera regla reconoce un secuencia no-vacia de tabuladores y espacios. Qué hacen las otras dos?

```
%{
#include <stdio.h>
void funciomeva(int);
}%
%option noyywrap
%%
[ \t]+          {printf(" ");}
\n              {;}
[0-9]+          {funciomeva(6);}
%%
void funciomeva(int a)
    {printf("%d",a/2);}
void main() {yylex(); }
```

## Ejercicios para pensar (en este orden) en casa y/o laboratorio

1. Escribe un programa Flex que copia un fichero, (i) reemplazando las secuencias de varios blancos por uno solo y (ii) que al acabar escriba el número de a's leídas.
2. Haz dos programas Flex, uno para encriptar y otro para desencriptar textos. La encriptación sustituirá cada carácter por otro que es función del carácter y de lo leído hasta el momento.
3. Tenemos un fichero de texto en el que puede haber en cualquier punto números en octal y hexadecimal. Escribe por pantalla el mismo texto, pero reemplazando los números en octal y hexadecimal por sus equivalentes en decimal. En la entrada, los números en octal comienzan por `0o` y en hexadecimal por `0x` (como en C, Haskell, CAML, y otros lenguajes de programación). Ejemplos:

octal/hexad.		decimal
0x100	=	256
0o100	=	64
0xabcd	=	43981
0o7654	=	4012

Hola0o100Yo soy Juan 0x100x100. Adios.  
daría:

Hola64Yo soy Juan 256x100. Adios.