



**Universidad
Europea de Madrid**

LAUREATE INTERNATIONAL UNIVERSITIES

ANÁLISIS LÉXICO

EXPRESIONES REGULARES

© Todos los derechos de propiedad intelectual de esta obra pertenecen en exclusiva a la Universidad Europea de Madrid, S.L.U. Queda terminantemente prohibida la reproducción, puesta a disposición del público y en general cualquier otra forma de explotación de toda o parte de la misma.

La utilización no autorizada de esta obra, así como los perjuicios ocasionados en los derechos de propiedad intelectual e industrial de la Universidad Europea de Madrid, S.L.U., darán lugar al ejercicio de las acciones que legalmente le correspondan y, en su caso, a las responsabilidades que de dicho ejercicio se deriven.

Índice

Presentación	4
Expresiones regulares	5
Operaciones con expresiones regulares	6
Definiciones regulares, precedencia y asociatividad	8
Implementación de un analizador léxico con Flex	9
Secciones de un programa Flex I	10
Secciones de un programa Flex II	12
Sección de Reglas	12
Secciones de un programa Flex III	14
Ejemplo de analizador léxico	15
Resumen	16

Presentación

El objetivo de este tema es aprender qué son las expresiones regulares, así como a operar con ellas. También aprenderemos a implementar un analizador léxico utilizando una herramienta generadora de analizadores léxicos, denominada **Flex**, de acceso libre.

Para ello seguiremos el siguiente índice:

- Expresiones regulares.
- Operaciones con expresiones regulares.
- Definiciones regulares, precedencia y asociatividad.
- Implementación de un analizador léxico con Flex.
- Secciones de un programa Flex.
- Ejemplo de analizador léxico.
- Errores más comunes con Flex.

Se probará a generar un analizador léxico básico con Flex, con la idea de ir ampliándolo de acuerdo con la gramática que se decida para el compilador.



Expresiones regulares

Se corresponden con las **gramáticas de tipo 3 de la jerarquía de Chomsky**. Las expresiones regulares son una forma de especificar **patrones**, entendiendo por patrón la forma de describir cadenas de caracteres. Es la forma de definir los **tokens o componentes léxicos** y, como veremos, cada patrón concuerda con una serie de cadenas.

De esta forma, utilizamos las expresiones regulares para darle nombre a estos patrones.

El lenguaje que se reconoce mediante estas expresiones regulares (**r**), se denomina **lenguaje generado por la expresión regular L(r)** (Louden, 2004).

Recordando la siguiente tabla:

Token (Componente léxico)	Lexema	Patrón
Identificador	a, valor, b	[a-zA-Z]+
Número	5, 3.25, 56	[0-9]+(\.[0-9]+)?

Vemos que para describir un identificador que se define solo por letras utilizando el patrón [a-zA-Z]+, que reconoce cualquier letra mayúscula o minúscula seguido del símbolo +, estamos indicando que al menos tiene que haber una letra para describir un identificador, pero no hay límite para el número de caracteres que puede tener ese token (identificador).

Una expresión regular se puede construir a partir de otras expresiones regulares más simples. Cuando definamos los símbolos mediante los que se especifican las expresiones regulares veremos ejemplos de estos, definiendo letras y dígitos y como un identificador es una combinación de ambos.

Para definir las expresiones regulares usamos **metacaracteres** o **metasímbolos** que especifican las acciones que se pueden reconocer sobre un determinado carácter o símbolo. Algunos de estos metacaracteres son: *, +, ?, |.

Cuando queremos utilizar estos símbolos, como por ejemplo * con su significado normal, para la operación de multiplicar se utiliza un carácter de escape que anula el significado especial del metacarácter. En este ejemplo la forma correcta sería *, y así tenemos el símbolo de multiplicar en un patrón.

Operaciones con expresiones regulares

Hay tres operaciones básicas con expresiones regulares:

Selección entre alternativas	Se denota por el metacarácter . Ejemplo: a b, significa que puede ser a ó b. Esta operación equivale a la unión, puesto que tanto a como b valdrían como lexemas para este patrón (obsérvese que utilizamos patrón y expresión regular de forma indistinta).
Concatenación	Se construye poniendo un símbolo al lado del otro y no utiliza ningún metacarácter. Ejemplo: ab, significa que el lexema equivalente tiene que ser "ab", sin alternativa posible.
Repetición	También se la denomina cerradura de Kleene y se denota por el metacarácter *. Identifica una concatenación de símbolos incluyendo la cadena vacía, es decir "0 o mas instancias" del símbolo afectado. Ejemplo: a*, significa que los lexemas para este patrón podrían ser: λ, a, aa, aaa, aaaa,...

Además de estas operaciones básicas, aparecieron posteriormente extensiones a las mismas para cubrir algunas lagunas a la hora de especificar patrones:

The diagram illustrates two extensions of basic operations, each shown with a smartphone displaying 'operaciones básicas'.

Extension 1: Positive Closure (+)

- Buttons: Una o más repeticiones (highlighted), Cero o una instancia, Intervalo de caracteres.
- Description: Se denota por el metacarácter +, también se le denomina cierre positivo. Esta operación indica que el símbolo afectado tendrá una o más instancias. Ej: a+, significa que los lexemas para esta expresión regular podrían ser: a, aa, aaa, aaaa, ...

Extension 2: Optional Character (?)

- Buttons: Una o más repeticiones, Cero o una instancia (highlighted), Intervalo de caracteres.
- Description: Se denota por el metacarácter ?, y significa cero o una ocurrencia del símbolo afectado. Ej: a?, significa que los lexemas que podrían valer son: λ ó a. Describe a un símbolo opcional.

Additional buttons for both extensions include: Clases de caracteres, Cualquier carácter, and Cualquier carácter que no pertenezca a un conjunto.

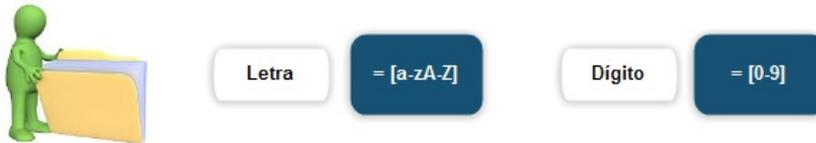
Una o más repeticiones	Cero o una instancia	Intervalo de caracteres
<p>Para especificar un intervalo de caracteres todos ellos válidos, podríamos usar la alternativa o utilizar los corchetes y un guion. Ej: [a-z], significa cualquier carácter de la a a la z, que también se podría haber especificado por a b c d y así sucesivamente hasta la z, pero esta forma es mucho mas abreviada y evita errores. También se pueden incluir los intervalos múltiples [a-zA-Z], que representa todas las letras minúsculas y mayúsculas.</p>		
Clases de caracteres	Cualquier carácter	Cualquier carácter que no pertenezca a un conjunto
Una o más repeticiones	Cero o una instancia	Intervalo de caracteres
<p>Es como el intervalo pero sin el guion y sirve también para abreviar las alternativas. Ej: [abc], equivale a a b c.</p>		
Clases de caracteres	Cualquier carácter	Cualquier carácter que no pertenezca a un conjunto
Una o más repeticiones	Cero o una instancia	Intervalo de caracteres
<p>Se denota por un punto ., Sirve para expresar que cualquier carácter encaja con la expresión regular. Se suele utilizar al final de la especificación de un analizador léxico por si queremos hacer alguna acción para todo lo que no concuerde con los patrones definidos.</p>		
Clases de caracteres	Cualquier carácter	Cualquier carácter que no pertenezca a un conjunto
Una o más repeticiones	Cero o una instancia	Intervalo de caracteres
<p>Se denota por ^, o también por la tilde ~, y significa cualquier carácter distinto a los que está afectando el metacarácter. Ej: [^a], significa cualquier carácter que no sea a.</p>		
Clases de caracteres	Cualquier carácter	Cualquier carácter que no pertenezca a un conjunto



Definiciones regulares, precedencia y asociatividad

Con el objeto de simplificar la **notación**, se pueden poner nombres a las expresiones regulares, con el objeto de no tener que volver a escribir la expresión cada vez que tenemos que utilizarla.

Por tanto, una definición regular es, a modo de ejemplo:



Pudiendo reutilizarlas para definir otro concepto, como por ejemplo:

- Identificador = letra (letra|dígito)*, donde queremos decir que un identificador una cadena que comienza por una letra a la que le pueden seguir 0 ó mas letras (minúsculas o mayúsculas) o dígitos. Ejemplo de lexemas serían: a, a3, a3b123. Los paréntesis sirven para indicar que todo lo que está incluido entre ellos se verá afectado por el metacarácter, en este caso el *, que implica 0 o mas repeticiones.

Otro aspecto importante de las expresiones regulares es la **precedencia y la asociatividad**. Está basada en la convención de que el orden de precedencia de mayor a menor es: **repetición (*, +, ?)**, **concatenación y alternativa** y todas son asociativas por la izquierda.

También es importante tener en cuenta los paréntesis, si queremos tener una precedencia diferente para determinados símbolos afectados por ellos. Como nos indica Louden (2004) en su ejemplo, (a|b) c, la operación alternativa tiene mayor precedencia que la concatenación al estar afectada por los paréntesis.

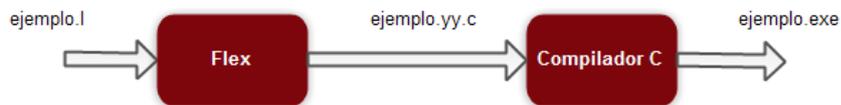
La norma es igual que en matemáticas, donde $(5 - 2) * 4 = 3 * 4 = 12$, la operación afectada por los paréntesis se realiza antes (tiene mayor precedencia) que la multiplicación, aunque sea una resta.

Implementación de un analizador léxico con Flex

Se han desarrollado herramientas para construir analizadores léxicos a partir de expresiones regulares. Un ejemplo de este tipo es **LEX** escrito por Mike Lex y Eric Schmidt, que en su versión libre se denominó Flex (Fast Lex).

Aprenderemos a utilizar las expresiones regulares, combinándolas con acciones para reconocer los símbolos de un lenguaje.

Posteriormente lo combinaremos con el analizador sintáctico para generar el embrión de un compilador.



El proceso es el siguiente:

Primero	Se escribe con un editor de texto (por ejemplo, Notepad) una especificación del analizador léxico mediante un programa denominado ejemplo.l, en lenguaje flex.
Segundo	Después ejemplo.l, se ejecuta con el flex para producir el programa en C, ejemplo.yy.c (flex ejemplo.l). Este programa construye una representación en forma de tabla de diagrama de transiciones que se construye a partir de las expresiones regulares definidas en ejemplo.l. Las acciones asociadas a las expresiones regulares de ejemplo.l se convierten en código escrito en C y se transfieren a ejemplo.yy.c.
Tercero	Compilamos ejemplo.yy.c con el compilador de C y obtenemos el programa ejecutable ejemplo.exe.

Secciones de un programa Flex I

Un programa en Flex tiene tres partes o secciones, separadas por una línea con `%%`.

La sección de declaraciones: son los `#include`, `#define`, declaración de variables (que serán globales), etc. que se copian tal cual en la cabecera del fichero "nombre_lex.lex.c". Son previas al uso del reconocimiento de símbolos.

Tiene cuatro partes:



Directivas de funcionamiento del flex

Cambian el funcionamiento por defecto de flex y las mas comunes son:

- %option noyywrap: permite usar flex sin necesidad de definir varios ficheros de entrada. Sirve para hacer varias pasadas. Cuando llega al final del fichero se llama a esta función que devuelve un 1 ó un 0, volviendo al principio del fichero para volver a tratarlo en función de ese valor.
- %option case-insensitive: considera las mayúsculas y minúsculas como el mismo carácter.
- %yylineno: permite guardar el número de línea que está procesando.

Código de usuario

Directivas de funcionamiento del flex

Definiciones regulares

Condiciones de arranque

Definiciones regulares

Permite dar nombre a patrones complejos, como por ejemplo, DIGITO [0-9].

Código de usuario

Directivas de funcionamiento del flex

Definiciones regulares

Condiciones de arranque

Condiciones de arranque

Permiten modificar el flujo de análisis y las hay de dos tipos:

- Inclusivas (%s variable): son evaluados los patrones con la condición de arranque y lo que no utilizan ninguna condición.
- Exclusivas (%x comentario): se evalúan solo los que cumplen la condición de arranque.

Código de usuario

Directivas de funcionamiento del flex

Definiciones regulares

Condiciones de arranque

Partes del programa en Flex

Declaraciones
%%
Reglas
%%
Código de usuario

Secciones de un programa Flex II

Sección de Reglas

Esta sección contiene reglas con el formato **PATRÓN ACCIÓN**, donde cada acción es un fragmento de programa que describe cual ha de ser la acción que realizará el analizador léxico cuando el patrón concuerde con un lexema del fichero que contiene el código fuente.

En los patrones se pueden utilizar **expresiones regulares**, **definiciones regulares** y condiciones de arranque. En el ejemplo que realizaremos en este tema el analizador léxico trabajará solo, pero normalmente trabaja de forma sincronizada con el analizador sintáctico, por tanto **ACCIÓN** devolverá el control al analizador sintáctico.

Las acciones son código C (es importante subrayar que todas las variables que se declaren antes de la primera regla, serán locales a la rutina que hace el escaneo.)

Una vez se reconoce un patrón o expresión regular, su valor está en **yytext**:

```
Ejemplo de patrón - acción: [abc] {printf ("%s", yytext); return(abc) ;}
```

¿Cómo se identifican los patrones cuando pueden aplicarse varias reglas?

1. Se aplica el patrón que concuerda con el número mayor de caracteres de la entrada
2. si hay dos patrones que concuerdan con el mismo número de caracteres en la entrada, entonces se aplica el que esté definido primero.



Ejemplo

En este caso entraremos por la última regla, puesto que es el patrón más largo que concuerda con la entrada, aplicando por tanto la regla 1. Flex incorpora una serie de variables para almacenar el lexema (yytext) y realizar acciones con sus valores (además de para comunicarse con el analizador sintáctico).

Condiciones de arranque

Las condiciones de arranque son un mecanismo que tiene Flex para activar reglas condicionalmente (véase Flex, 1995).

Variables que incorpora Flex

Variable	Tipo	Descripción
yytext	char* o char []	Contiene la cadena de texto del fichero de entrada que ha encajado con la expresión regular descrita en la regla.
yytext	int	Longitud de yytext Yylength = strlen (yytext)
yyin	FILE*	Referencia al fichero de entrada
Yyval yyval	struct	Contienen la estructura de datos de la pila con la que trabaja Bison o Yacc (son los generadores de analizadores sintácticos). Sirve para el intercambio de información entre ambas herramientas.

Ejemplo

Entradas

Tenemos la siguiente entrada **xyz**:

x	{return(1); }
xy	{return(2); }
z	{return(3); }
xyz	{return(4); }

Secciones de un programa Flex III

Sección de código de usuario: el código que se escribe en esta sección se traslada de forma íntegra al fichero nombre_flex.yy.c.

Esta sección es opcional y si no se escribe es equivalente a escribir lo siguiente:

```
1. int main ()
2. {
3.     yyin = stdin;
4.     yylex();
5. }
6. Ejemplo de main que acepta un fichero como entrada:
7. int main (int argc, char *argv[])
8. {
9.     if (argc == 2)
10.    {
11.        yyin = fopen (argv[1], "rt");
12.        if (yyin == NULL)
13.        {
14.            printf ("Fichero %s erróneo\n", argv[1]);
15.            exit (-1);
16.        }
17.    }
18.    else yyin = stdin;
19.    yylex ();
20.    return 0;
21. }
```

Acepta la entrada estándar, a través del teclado y los caracteres que se van escribiendo los pasa al reconocedor

En este caso la entrada es un fichero que se pasa como argumento y mientras tenga caracteres (yyin distinto de NULL) sigue pasándose al analizador léxico que hemos generado

Ejemplo de analizador léxico

```

1.  %{
2.  #include <stdio.h>
3.  %}
4.  digito [0-9]
5.  letra [a-zA-Z]
6.  %%
7.  "if" {printf("IF\n");}
8.  "else" {printf("ELSE\n");}
9.  {letra}{letra}|{digito})* {printf("IDENTIFICADOR: %s\n",yytext);}
10. . ; /*no hace nada*/
11. %%

```

Creamos unas definiciones regulares, como dígito y letra que son aprovechadas para reconocer un identificador. Además reconocemos un par de palabras reservadas (if y else). El punto indicará que cualquier otro carácter será descartado

Como hemos indicado anteriormente, si en la sección de código de usuario no se escribe nada, acepta los caracteres de la entrada por teclado

Si no le ponemos nada más permitirá que salga por la salida estándar (pantalla) y acepte caracteres por la entrada estándar (teclado).

Una vez funcione, se le puede incorporar el código de usuario visto anteriormente que permite que se le pueda pasar un fichero con el código fuente a reconocer. El proceso de compilación es el siguiente:

Primero	Editamos ejemplo.l, con el código indicado en esta pantalla (para posteriormente ir ampliando).
Segundo	flex ejemplo.l, obtendremos ejemplo.yy.c
Tercero	gcc -o ejemplo.exe ejemplo.yy.c

Resumen

En este tema hemos entendido qué son y para qué sirven las expresiones regulares, cómo funciona un analizador léxico y cómo se relaciona con el analizador sintáctico y con otras estructuras necesarias como la tabla de símbolos. Para hacernos una idea más completa hemos visto todas las funciones que debe llevar a cabo el analizador léxico con el fichero de entrada que contiene el código fuente. Además hemos conocido que ventajas aporta a un compilador.

Más adelante hemos comprendido cuáles son los conceptos básicos (token o componente léxico, lexema y patrón), donde hemos aprendido que un token puede tener uno (palabra reservada u operadores) o infinitos (identificadores o números) lexemas.

Por otro lado, hemos entendido cómo funciona el analizador léxico y cómo se construye a partir de una expresión regular el autómata finito que lo soporta. Estos conceptos los trataremos en mayor detalle en los dos próximos temas.

También se ha conocido cómo se diseña un analizador léxico por medio de una tabla o un diagrama de transiciones, representando de esta forma los estados por los que pasa el analizador para reconocer un token.

Posteriormente hemos aprendido mediante un ejemplo a reconocer un identificador, y esto puede extenderse a los números enteros, números decimales, operadores, comentarios, etc. Además se han identificado tres formas de implementar estos ejemplos; generador de analizadores léxicos, lenguaje de alto nivel y lenguaje de bajo nivel.

Para finalizar hemos visto los errores léxicos que se pueden detectar y la forma de tratarlos para eliminar esos errores o minimizar su impacto en el resto del compilador.