Universidad de La Laguna

Grado en Ingeniería Informática

Computabilidad y Algoritmia

PRÁCTICA 4

Expresiones regulares

Factor de ponderación: 7

1. Objetivos

El objetivo de la práctica es trabajar con expresiones regulares en C++. En el ámbito de la teoría de autómatas y los lenguajes formales, una expresión regular (*regular expression* o *regex*) es un mecanismo formal que permite describir lenguajes regulares. Es decir, toda expresión regular representa a un lenguaje regular. Desde un punto de vista más general podríamos decir que una expresión regular es una secuencia de caracteres que conforma un patrón de búsqueda. Las expresiones regulares proporcionan una manera muy flexible de buscar o reconocer cadenas en un texto.

Los principales lenguajes de programación tienen soporte para la utilización de expresiones regulares (C++, Python, Java, Ruby, etc.). Esto se debe a que habitualmente los programadores necesitan trabajar con cadenas de texto (strings) para validar los datos ingresados por el usuario, validar formatos de una URL, de un correo electrónico, reemplazar palabras en párrafos, etc.

A modo de resumen, podríamos decir que los usos principales de las expresiones regulares son los siguientes [2]:

- **Buscar** elementos particulares (determinadas subcadenas) dentro de un texto principal (cadena). Por ejemplo, es posible identificar todas las direcciones de correo electrónico presentes en un archivo.
- **Reemplazar** elementos particulares (determinadas subcadenas) dentro de un texto principal (cadena). Por ejemplo, es posible querer eliminar todos los comentarios de un fichero de código fuente.

- **Validar** entradas comprobando que una determinada cadena cumple con un patrón específico. Por ejemplo, puedes querer verificar que una contraseña cumpla con ciertos criterios, tales como: una combinación de mayúsculas y minúsculas, dígitos, signos de puntuación, etc.
- **Seleccionar** elementos en base a ciertos patrones. Por ejemplo, es posible si quieres procesar ciertos archivos en un directorio, pero solo si cumplen condiciones particulares.
- **Formatear** contenido en un archivo. Por ejemplo, puedes querer mejorar el estilo de tu código asegurándote de que entre cualquier operador binario hay espacio en blanco tanto a la izquierda como a la derecha.

Tal y como ya hemos mencionado, la librería estándar de C++ proporciona soporte para el uso de expresiones regulares a través del módulo <**regex**>. El objetivo de esta práctica será precisamente aprender a utilizar expresiones regulares en C++ a través de las clases y las funcionalidades proporcionadas en <**regex**>. Al igual que en las prácticas anteriores, también se propone que el alumnado utilice este ejercicio para poner en práctica aspectos generales relacionados con el desarrollo de programas en C++. Algunos de estos principios que enfatizaremos en esta práctica serán los siguientes:

- **Programación orientada a objetos**: es fundamental identificar y definir clases y objetos que permitan modelar adecuadamente el escenario de trabajo que se plantea.
- **Diseño modular**: el programa debiera escribirse de modo que las diferentes funcionalidades que se precisen sean encapsuladas en métodos concretos cuya extensión textual se mantenga acotada.
- **Pautas de estilo**: es imprescindible ceñirse al formato para la escritura de programas en C++ que se propone en esta asignatura [11]. Algunos de los principales criterios de estilo ya se han descrito reiteradamente en las prácticas anteriores de esta asignatura.
- **Expresiones regulares**: se utilizará la librería <**regex**> para hacer uso de expresiones regulares en programas escritos en C++.
- Compilación de programas utilizando `make` [9, 10].

2. Regex

La librería estándar de C++ suministra soporte para expresiones regulares a través de la librería de expresiones regulares (desde C++11). Para utilizar esa librería es necesario incluir en los programas C++ el fichero de cabecera `<regex>`.

Para comenzar a utilizar expresiones regulares en C++ es importante conocer las operaciones básicas proporcionadas por esta librería:

- `regex_match`: devuelve si la secuencia de caracteres completa coincidió con la expresión regular especificada, opcionalmente capturando en un objeto coincidente.
- `regex_search`: devuelve si una parte de la secuencia de caracteres coincidió con la expresión regular, opcionalmente capturando en un objeto coincidente.
- `regex_replace`: devuelve la secuencia de caracteres de entrada modificada por una expresión regular a través de una cadena de formato de reemplazo.

Tanto `regex_match` como `regex_search` son funciones coincidentes. La principal diferencia es que `regex_match` comprueba si una expresión regular coincide con **toda** la cadena de destino mientras que `regex_search` se detiene cuando encuentra la primera coincidencia y no realiza múltiples búsquedas. Por su parte, `regex_replace` realiza múltiples búsquedas, reemplazando la cadena coincidente encontrada con la cadena de reemplazamiento especificada.

A la hora de llevar a cabo estas operaciones de búsqueda y/o reemplazamiento, intervienen los elementos siguientes:

- *Target sequence*: secuencia destino o *string* sobre el cual se realizará la búsqueda del patrón.
- *Regular expression*: expresión regular que define el patrón a encontrar en la secuencia destino.
- *Matched array*: estructura en la que se almacena la información sobre el *matching* o emparejamiento realizado.
- *Replacement string*: cadena de reemplazo que sustituirá al patrón especificado dentro de la secuencia destino.

Para poder especificar el patrón de búsqueda se utiliza una sintaxis de expresiones regulares que, además de incluir los operadores básicos de unión, concatenación y asterisco, admite extensiones de dichos operadores. Algunos de estos operadores ya se introdujeron en clase, pero existen algunos otros que además de ser válidos en la librería `regex` de C++ también lo son en algunas de las librerías disponibles en otros lenguajes de programación.

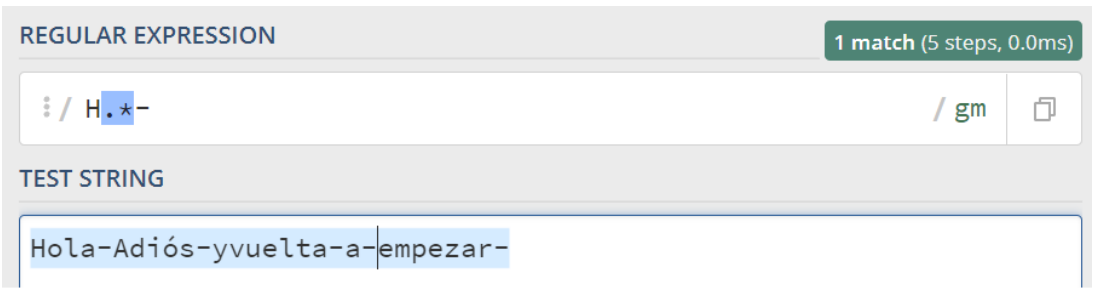
A continuación se relacionan los principales operadores extendidos que se suelen utilizar para definir expresiones regulares:

$[A-Z]$	Cualquier letra mayúscula
$[0-9]$	Cualquier dígito
$[A-Za-z0-9]$	Cualquier letra o dígito
.	Cualquier caracter (excepto el salto de línea)
+	Una o más ocurrencias
*	Cero o más ocurrencias
?	Cero o una única ocurrencia
{m}	Exactamente m ocurrencias
{m,n}	Entre m y n ocurrencias (ambas incluidas)
|	Una cosa u otra (`or`)
^	Primera ocurrencia (comienzo del patrón)
$	Última ocurrencia (final del patrón)
\+, *, \\	Los caracteres especiales hay que antecederlos de \
\d	Cualquier dígito
\D	Cualquier caracter que no sea un dígito
\w	Cualquier letra o dígito
\W	Cualquier caracter que no sea letra ni dígito
\s	Cualquier blanco
\S	Cualquier caracter que no sea un blanco
( )	Captura de grupos y/o subgrupos

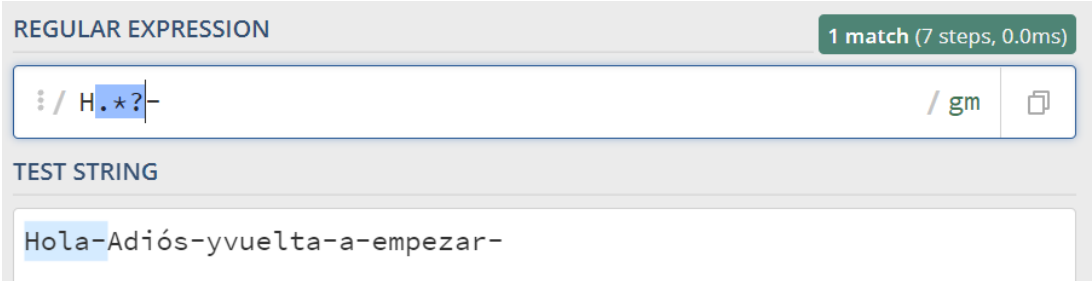
Para entender mejor y poner en práctica la sintaxis de las expresiones regulares comience por estudiar el tutorial interactivo disponible en [4]. También se recomienda la lectura de los materiales disponibles en [2, 5].

En lo que se refiere a la sintaxis, es importante hacer inciso en dos aspectos: el comportamiento "*greedy*" del operador `*` y la utilización de los agrupamientos para luego acceder a partes concretas de la secuencia coincidente.

En primer lugar, es importante tener en cuenta que los cuantificadores de repetición por defecto tienen un comportamiento ávido (*greedy*). Para mostrar este comportamiento se ha utilizado una herramienta para el testeo online con expresiones regulares [8]. Si nos fijamos en la expresión regular siguiente (cadenas que empiezan por H seguida de cero o más caracteres y terminan en guión), veremos que, por defecto, el `.*` casará con toda la cadena, llegando hasta el último guión (-). A este comportamiento se le llama ávido o *greedy* porque casa con la mayor cadena posible [6].



Sin embargo, si utilizáramos **.*?** estaríamos usando un cuantificador perezoso o *lazy*. De esta forma, al realizar el emparejamiento se pararía con la primera subcadena (subcadena más corta) que cumpla con el patrón especificado [7].



Por otro lado, la utilización de grupos sirve para capturar partes de la subcadena que ha casado con el patrón de búsqueda. Tomando como ejemplo el código siguiente:

```
#include <iostream>
#include <regex>
#include <string>

using namespace std;

int main () {
  // Target sequence
  string target_s = "I am looking for GeeksForGeeks articles";

  // An object of regex for pattern to be searched
  regex expression("Geeks[a-zA-Z]+" );

  // flag type for determining the matching behavior
  // here it is for matches on 'string' objects
  smatch matches;

  // regex_search() for searching the regex pattern
  // 'expression' in the string 'target_s'.
  // 'matches' is flag for determining matching behavior.
  regex_search(target_s, matches, expression);
  for (auto match: matches)
    cout << match << std::endl;
  return 0;
}
```

En el array `matches` se almacena la información de las coincidencias (*matchings*) obtenidas. En este caso, como no se han usado agrupamientos, la única cadena mostrada en el bucle sería:

```
GeeksForGeeks
```

Si, por el contrario, en la expresión regular se hubiera usado un agrupamiento para "almacenar" la cadena que va entre *Geeks* y el primer caracter diferente de una letra:

```
regex r("Geeks([a-zA-Z]+)")
```

Entonces el bucle habría mostrado dos cadenas, la cadena completa del emparejamiento y aquella parte que ha casado con el agrupamiento especificado:

```
GeeksForGeeks
ForGeeks
```

3. Ejercicio práctico

Para poner en práctica lo estudiado hasta el momento, se desarrollará un programa en C++ que analice el contenido de ficheros de código haciendo uso de expresiones regulares. El programa recibirá por línea de comandos el nombre del fichero de entrada y el nombre del fichero de salida:

```
./p04_code_analyzer code.cc codescheme.txt
```

En caso de ejecutarse sin parámetros, el programa mostrará un mensaje de error indicando el modo correcto de ejecución, indicando los parámetros necesarios y el significado de cada uno.

El fichero de entrada será un fichero de código C++ sintácticamente correcto. El fichero de salida resumirá la estructura del fichero de entrada: variables declaradas, bucles utilizados, comentarios incluidos así como indicación de la existencia o no de un programa principal. El fichero de salida no se creará directamente mientras se analiza el fichero de entrada sino que **se creará al menos una clase que represente y almacene la estructura general del código** C++ analizado.

Se deberá hacer uso de expresiones regulares para analizar el código fuente de entrada y extraer la información siguiente:

- **Declaración de variables**: se detectarán las declaraciones de variables de tipo `int` así como las de tipo `double`. Se supondrá que cada declaración de variable se hará en una única línea y que en cada línea tendremos una única declaración. Por lo tanto, no se declararán dos o más variables en una misma línea ni se considerarán aquellas variables declaradas dentro de bloques como bucles `for` pues no se encuentran declaradas en una línea independiente. Para cada variable se almacenará, además de su tipo, su nombre, la línea en la que se declara y si se ha inicializado o no en la propia declaración. Se llevará un control de cuántas variables de cada tipo se han definido en el programa.

- **Utilización de bucles**: se detectarán los bucles de tipo `for` y los bucles de tipo `while`. Para cada bucle detectado se almacenará el tipo de bucle y la línea del código en la que se ha encontrado. Además, se llevará un control del número de bucles de cada tipo que se han utilizado en el programa.
- **Programa principal**: se detectará si existe o no una función `main` en el código fuente analizado.
- **Comentarios**: se detectarán todos los comentarios de una línea (`//`) y de múltiples líneas (`/* */`) presentes en el fichero. Se almacenará el tipo de cada comentario, la/s línea/s en la que se encuentra cada uno así como el contenido del propio comentario. Además, se tendrá en cuenta que si al comienzo del fichero se encuentra un comentario, ese comentario se tomará por defecto como descripción del programa.

A modo de ejemplo, si se considera el código de entrada siguiente:

```
/**
 * Universidad de La Laguna
 * Escuela Superior de Ingenieria y Tecnologia
 * Informatica Basica
 *
 * @author F. de Sande
 * @date 23.nov.2020
 * @brief Ejercicios Informatica Basica
 *        Version 1: Funcion factorial
 *
 * @see https://github.com/IB-2022-2023/IB-class-code-examples/
 */

#include <iostream>
#include <cassert>

// Returns the factorial of the argument
int Factorial(int number) {
  switch (number) {
    case 0:
    case 1:
      return 1;
    default:
      int factorial = 1;
      for (int i = 1; i <= number; ++i) {
        factorial *= i;
      }
      return factorial;
  }
}

int main () {
  std::cout << "Introduzca el numero de factoriales a calcular: ";
  int limit;
  std::cin >> limit;
  for (int i = 1; i <= limit; ++i) {
```

```
37      std::cout << i << "! = " << (double)Factorial(i) << std::endl;
38    }
39    return 0;
40  }
```

La salida del programa debería ser un fichero similar al siguiente:

```
PROGRAM: factorial.cc
DESCRIPTION:
/**
 * Universidad de La Laguna
 * Escuela Superior de Ingenieria y Tecnologia
 * Informatica Basica
 *
 * @author F. de Sande
 * @date 23.nov.2020
 * @brief Ejercicios Informatica Basica
 *        Version 1: Funcion factorial
 *
 * @see https://github.com/IB-2022-2023/IB-class-code-examples/
 */

VARIABLES:
[Line 30] INT: factorial = 1
[Line 40] INT: limit

STATEMENTS:
[Line 31] LOOP: for
[Line 42] LOOP: for

MAIN:
True

COMMENTS:
[Line 1-18] DESCRIPTION
[Line 23] // Returns the factorial of the argument
```

Para más ejemplos de códigos en C++ que podría utilizar para testear su programa, le sugerimos revisar el repositorio de la asignatura de Informática Básica:
https://github.com/IB-2022-2023/IB-class-code-examples/

4. Criterios de evaluación

Se señalan a continuación los aspectos más relevantes (la lista no es exhaustiva) que se tendrán en cuenta a la hora de evaluar esta práctica:

- Se valorará que el alumnado haya realizado, con anterioridad a la sesión de prácticas, y de forma efectiva, todas las tareas propuestas en este guión. Esto implicará que el programa compile y ejecute correctamente.
- También se valorará que, con anterioridad a la sesión de prácticas, el alumnado haya revisado los documentos que se enlazan desde este guión.
- Paradigma de programación orientada a objetos: se valorará que el alumnado haya identificado clases y objetos que permitan modelar adecuadamente el escenario de trabajo que se planeta.
- Paradigma de modularidad: se valorará que el programa se haya escrito de modo que las diferentes funcionalidades que se precisen hayan sido encapsuladas en métodos concretos cuya extensión textual se mantuviera acotada.
- Uso de expresiones regulares a través de la librería `regex`: se valorará el uso de expresiones regulares y de operaciones básicas como `match`, `search` o `replace`.
- Se valorará que el código desarrollado siga el formato propuesto en esta asignatura para la escritura de programas en C++.
- Capacidad del programador(a) de introducir cambios en el programa desarrollado.

Si el alumnado tiene dudas respecto a cualquiera de estos aspectos, debiera acudir al foro de discusiones de la asignatura para plantearlas allí. Se espera que, a través de ese foro, el alumnado intercambie experiencias y conocimientos, ayudándose mutuamente a resolver dichas dudas. También el profesorado de la asignatura intervendrá en las discusiones que pudieran suscitarse, si fuera necesario.

Referencias

[1] Transparencias del Tema 2 de la asignatura: Autómatas finitos y lenguajes regulares, https://campusingenieriaytecnologia2223.ull.es/mod/resource/view.php?id=5919

[2] Tutorial de Regex: Aprender con ejemplos de expresiones regulares, https://content.breatheco.de/es/lesson/regex-tutorial-regular-expression-examples

[3] The Regex Library, http://www.cplusplus.com/reference/regex

[4] RegexOne: Learn Regular Expressions with simple, interactive exercises, https://regexone.com

[5] Regular Expression Basics in C++: Patterns, https://linuxhint.com/regular-expression-basics-cpp/#a2

[6] Why Using the Greedy .* in Regular Expressions Is Almost Never What You Actually Want, https://mariusschulz.com/blog/why-using-the-greedy-in-regular-expressions-is-almost-never-what-you-actually-

[7] Expresiones regulares: cuantificadores perezosos, https://learntutorials.net/es/cplusplus/topic/1681/expresiones-regulares

[8] Interactive Regular Expressions, https://regex101.com

[9] Makefile Tutorial: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor

[10] C++ Makefile Tutorial: https://makefiletutorial.com

[11] Google C++ Style Guide, https://google.github.io/styleguide/cppguide.html