Autómatas y Lenguajes Formales

Boletín de Prácticas

Curso 2023-2024

UNIVERSIDAD
DE MURCIA

TABLA DE CONTENIDOS

Proyecto de programación

Las prácticas de la asignatura Autómatas y lenguajes Formales se realizan por parejas y consisten en el desarrollo de una aplicación en lenguaje Python que haga uso de Expresiones Regulares. A lo largo de las sesiones de prácticas presenciales se trabajarán todos los aspectos necesarios para poder desarrollar dicha aplicación.

Además del código, se deberá escribir y entregar un informe describiendo el trabajo realizado, que constará de los siguientes apartados:

- **Portada** con la siguiente información: Apellidos, Nombres, DNIs, Asignatura, Grupo y Subgrupo de Prácticas, Convocatoria, Año académico y Profesor.
- **Tabla de contenidos**.
- **Manual de usuario**: Instrucciones para utilizar la aplicación.
- **Aspectos principales**: Explica cómo has resuelto los distintos problemas usando expresiones regulares y los aspectos de programación más relevantes: TDA's, etc.
- **Conclusiones**: Comentarios personales con relación a las dificultades encontradas, la forma de superarlas y el grado de satisfacción con la asignatura, así como sugerencias de mejora.

El texto del informe se debe escribir usando una fuente similar a **Arial** tamaño 11. En caso de incluir segmentos de **código fuente** en el informe estos deberán estar bien formateados. Es decir, se debe cuidar la indentación y usar un tamaño de la letra adecuado para que no se corten los renglones, o que cuando esto ocurra el código siga siendo legible y ordenado. El tipo de letra para **el código fuente** será `courier` o uno semejante de ancho fijo.

Entregables

Todo el material se entregará **comprimido en un único fichero en formato ZIP cuyo nombre será una lista de los DNIs de sus autores separados por guiones (-) y cuyo contenido estará distribuido en carpetas** del siguiente modo:

- **Carpeta "*code*"**: Contendrá únicamente los ficheros siguientes:
    - Ficheros de código fuente.
    - Otros ficheros necesarios para compilar o ejecutar la aplicación tales como imágenes, textos, datos, etc.
- **Carpeta "*doc*"**: Contendrá el informe en formato PDF.

IMPORTANTE: El código debe compilar y ejecutar sin avisos (warnings) ni errores de compilación en unas condiciones similares a las de los laboratorios de prácticas.

Fecha y lugar de entrega

La entrega se realizará a través de una tarea publicada en el Aula Virtual con suficiente antelación. Sólo debe hacer la entrega uno de los miembros del grupo.

Criterios de evaluación

Si el profesor lo considerase oportuno se pedirá a los estudiantes la realización de una entrevista personal para aclarar detalles del trabajo realizado. En tal caso, la no asistencia (injustificada) a esta entrevista de prácticas será motivo de suspenso.

El proyecto de programación será valorado por el profesor con una puntuación de 0 a 10. Para superar la parte práctica de la asignatura será necesario obtener una nota igual o superior a 5.

En la evaluación se tendrá en cuenta la presentación, claridad y completitud del informe, el uso de un lenguaje técnico apropiado, la legibilidad, organización y documentación del código fuente, y la corrección, robustez, modularidad, extensibilidad, reusabilidad y eficiencia del software.

A continuación, se muestran algunos ejemplos de aspectos relevantes que se tendrán en cuenta durante la evaluación del software:

1. Legibilidad
   - Elige los identificadores de variables y funciones de forma adecuada.
   - Documenta las funciones incluyendo precondiciones, efecto y resultados.
   - Formatea el código fuente aplicando sangrado.
2. Corrección
   - Evita la generación de errores en tiempo de ejecución.
   - Asegura la corrección de los datos introducidos por el usuario.
   - Evita la aparición de bucles infinitos y llamadas recursivas que desborden la pila.
3. Robustez
   - Da valor inicial a las variables.
   - Controla errores (excepciones) producidos en tiempo de ejecución.
   - Comprueba los valores devueltos por funciones que informen sobre errores.
4. Modularidad
   - Aplica abstracción procedimental creando funciones para no repetir código.
   - Distribuye el código en módulos de forma coherente.
5. Extensibilidad y Reusabilidad
   - Crea constantes en lugar de usar valores literales en el código.
   - Evita el uso de variables globales.
   - Parametriza las funciones y módulos para hacerlos más generales.
6. Eficiencia
   - Selecciona la representación más adecuada para las estructuras de datos.
   - Utiliza esquemas de inserción, acceso, recorrido y búsqueda eficientes y adecuados.
   - Evita la repetición innecesaria de cálculos o acciones previamente realizadas.

Descripción del trabajo a realizar

La aplicación a realizar consiste fundamentalmente en un silabeador, es decir, un programa capaz de separar en sílabas las palabras de un texto en castellano. Usando dicha información, la aplicación podrá realizar otras tareas como poner guiones ortográficos para dividir una palabra al final de una frase cuando se formatea un texto, o encontrar palabras que rimen.

Concretamente, la aplicación debe tener un bucle interactivo en el que el usuario pueda decidir qué hacer entre las siguientes opciones:

- Silabear una palabra
- Clasificar una palabra según su entonación
- Obtener palabras que rimen con una dada
- Justificar un texto
- Salir

El programa contará con una BBDD representada mediante un fichero en formato CSV dónde se guardarán todas las palabras que el usuario vaya introduciendo junto con la información sobre división en sílabas y entonación. Cada vez que el programa arranque leerá dicho fichero y al terminar lo sobrescribirá con la información que tenga en memoria.

Las dos primeras opciones, junto con la de salir, son obligatorias mientras que las dos últimas son opcionales. Las funciones para silabear y determinar la entonación de una palabra deben resolverse utilizando expresiones regulares usando los métodos del paquete de expresiones regulares.

Haciendo la parte obligatoria se puede obtener una nota máxima de 7 puntos y haciendo las opcionales hasta los 10. La descripción concreta de los dos ejercicios opcionales está en los anexos.

A lo largo de las sesiones guiadas se trabajarán todos los aspectos necesarios para poder desarrollar la aplicación.

Sesiones de prácticas guiadas

Sesión 1. Introducción a Python I

1) Escribe un programa que muestre el número de segundos que hay en 2000 años
2) Escribe un programa que muestre si el número 7 es par o no
3) Escribe un programa que muestre si 997 es primo o no
4) Escribe una función llamada primo que devuelva verdadero si el único parámetro que debe tener la misma es un número primo
5) Escribe un programa que use la función anterior para mostrar los primeros 1000 números primos

Sesión 2. Introducción a Python II

1) Escribe el código de la función siguiente

   ```
   entonación( lista de sílabas )
   ```

   Esta función debe detectar si la palabra que representa la lista de sílabas recibida como parámetro es aguda, llana, esdrújula o sobreesdrújula y cuál es la vocal tónica. Para hacerlo se usarán las reglas que se pueden encontrar en el anexo 2. El resultado será una nueva lista de sílabas "anotada" en la que la vocal tónica estará en mayúscula. Por ejemplo: de ["a", "ba", "bol"] se obtendrá ["a", "ba", "bOl"], de ["cui", "dar"] se obtendrá ["cui", "dAr"] y de ["a", "güe", "ro"] se obtendrá ["a", "güE", "ro"] y de ["ca", "mión"] se obtendrá ["ca", "miOn"]. Fíjate que las tildes desaparecen. Esto facilitará algunas operaciones posteriores de comparación.

Sesión 3. Introducción a Python III

1) Escribe un programa que pida al usuario una palabra. Tras introducirla el programa mostrará el número de letras y si tiene alguna mayúscula. Si la respuesta es una línea vacía el programa terminará.
2) Amplía el programa para que tenga un menú en el que haya dos opciones. La primera hará lo mismo que el ejercicio anterior, la segunda hará el mismo análisis pero con cada una de las palabras de un fichero de texto cuyo nombre se pedirá al usuario.
3) Amplía el programa para que guarde el resultado del análisis en otro fichero en formato csv. Cada renglón tendrá una palabra junto con su longitud como segundo campo y un tercer campo con un valor distinto de cero si contiene mayúsculas o cero si no.

Sesión 4. Expresiones Regulares con Sintaxis Extendida I

1) Diseña expresiones regulares para detectar y contar apariciones no solapadas de cada uno de los patrones que definen las reglas de división silábica. Escribe una ER para cada caso.
2) Prueba las expresiones con el fichero Vocabulario.txt en EditPad-Pro. Recuerda activar el botón de "resaltar", la opción "regex" y activar la opción "case sensitive". Observa que va a dejar apariciones sin detectar pues esta herramienta sólo detecta las **apariciones no solapadas**. Observa también que habrá cadenas que **casen con varias reglas**. De momento, no nos preocupamos de estos problemas.
3) Crea una expresión regular con **la unión de todas las anteriores**. Elige bien el orden de cada elemento de la unión ya que en esta implementación **no es un operador conmutativo**. Usa los paréntesis cuando sea necesario. Pruébala en EditPad-Pro.

Sesión 5. Expresiones Regulares con Sintaxis Extendida II

1. Modifica las expresiones regulares de la sesión anterior para
    a. Ignorar la capitalización, es decir las mayúsculas al inicio de palabra.
    b. Incorporar **nombres de grupos** correspondientes a cada una de las reglas y, dentro de cada regla, nombres de grupo correspondientes al final de la sílaba anterior y al comienzo de la siguiente.
2. Comprueba con Editpad-Pro que funcionan usando la operación de "resaltar todas". Recuerda que sólo detecta apariciones no solapadas pero, de momento, no nos preocupamos.
3. Usa Editpad-Pro para reemplazar cada palabra del fichero Vocabulario.txt por su correspondiente palabra silabeada (es decir, con sus sílabas separadas por un guión). Para ello utilizaremos la operación "reemplazar" o "reemplazar todas". Como sólo detecta apariciones no solapadas, habrá sílabas que no queden separadas en la primera aplicación de las reglas pero puedes aplicarla más veces.

Sesión 6. Introducción a Python IV

1. Escribe el código de la función siguiente:

    `silabear( cadena )`

    Esta función debe devolver una lista de sílabas correspondiente a la palabra recibida en el parámetro cadena. Cada sílaba será, por tanto, una subcadena de la original. Por ejemplo: de "camión" se obtendrá ["ca", "mión"], y de "vivíais" se obtendrá ["vi", "ví", "ais"]. Fíjate que es importante conservar las tildes.

    Para hacer esta función se utilizarán las expresiones regulares obtenidas en sesiones anteriores. Además, siempre se asumirá que la ortografía de la palabra proporcionada a la función es correcta.
2. Modifica el programa de la sesión 3 sustituyendo la primera opción por la de silabear una palabra.

Sesión 7. Introducción al Python V

1) Escribe una nueva función `entonación`. A diferencia del ejercicio de la sesión 2, esta vez utiliza expresiones regulares para resolver el problema.
2) Para hacer la aplicación más interesante y eficiente vamos a añadirle memoria y persistencia. En concreto debes usar un diccionario para almacenar cada palabra sobre la que se realice cualquier tipo de análisis a petición del usuario. Las palabras serán las claves del diccionario y como valor guardaremos el resultado del análisis. De este modo, si vuelve a necesitarse ya estará calculado. Así pues, una opción conveniente es usar como valor otro diccionario con toda la información que podamos obtener sobre una palabra: su lista de sílabas, su entonación y, en su caso, sus rimas (ampliaciones).
   Cada vez que el programa termine guardará el diccionario en un fichero concreto en formato CSV, y cada vez que arranque comprobará si dicho fichero existe y, de existir, lo leerá creando el diccionario con el que empezará a funcionar.

Anexo 1. La División Silábica

Estructura General de una Sílaba en Castellano

La estructura general de una **sílaba** en el español es la siguiente:

SÍLABA						
ATAQUE		NÚCLEO			CODA	
C	C	V	V	V	C	C

donde C es una **consonante** y V es una **vocal**. Se considerará que son consonantes del español, a efectos fonéticos, no sólo las conocidas grafías individuales: b, c, d, f, g, h, j, k, l, m, n, ñ, p, q, r, s, t, v, w, x, y, z sino también los **grupos consonánticos** ch, ll y rr. Las vocales son a, e, i, o, u, sus versiones acentuadas á, é, í, ó, ú, ü, junto con y, que sólo se puede usar como vocal al final de palabra (convoy, Paraguay).

El único componente obligatorio de la sílaba es el **núcleo**, y en concreto la vocal central, que se denomina **vocal silábica**. Las **vocales no silábicas** (opcionales) aparecen en diptongos y triptongos, como se verá más adelante. Cualquiera de las cinco vocales a, e, i, o, u, así como sus versiones acentuadas, puede hacer la función de vocal silábica. Las **vocales no silábicas** son i, u e y al final de palabra.

Normas de división silábica

La siguiente tabla muestra, en la parte superior, dos sílabas seguidas, con todas las vocales y consonantes posibles en cada una de ellas. Hay que considerar que muchas palabras del español no emplean todas las vocales y consonantes posibles en cada sílaba. Por eso, en las filas sucesivas se muestran las distintas combinaciones que habrá que considerar para detectar el **límite silábico** entre cualquier par de sílabas.

La estrategia para encontrar las sílabas consiste en buscar patrones adecuados que identifiquen la porción de letras que va desde el final del núcleo de una sílaba hasta el comienzo del núcleo de la siguiente, es decir, entre dos vocales. No obstante, también habrá ciertos casos en los que será necesario comprobar si dos o tres vocales van unidas en la misma sílaba (diptongos y triptongos).

		SILABA 1							SILABA 2						
		ATAQUE		NÚCLEO			CODA		ATAQUE		NÚCLEO			CODA	
REGLA	*EJEMPLO*	C	C	V	V	V	C	C	C	C	V	V	V	C	C
1.	u-no					V_1	-	-	C	-	V_2				
2.a.	i-glú					V_1	-	-	C_1	C_2	V_2				
2.b.	o-tro					V_1	-	-	C_1	C_2	V_2				
2.c.	is-la					V_1	C_1	-	C_2	-	V_2				
3.a.	des-gracia					V_1	C_1	-	C_2	C_3	V_2				
3.b.	ins-tar					V_1	C_1	C_2	C_3	-	V_2				
3.c.	ist-mo					V_1	C_1	C_2	C_3	-	V_2				
4.	ins-crito					V_1	C_1	C_2	C_3	C_4	V_2				
5.a.	ciudad				V_1	V_2									
5.b.	ra-íz					V_1	-	-	-	-	V_2				
6.a.	buey			V_1	V_2	V_3									

Al aplicar las normas que aparecen en las siguientes subsecciones se puede determinar la situación del comienzo del ataque de cada sílaba. Obsérvese que el ataque de la primera sílaba no se contempla en las siguientes reglas, porque es evidente que comienza en la primera posición de la palabra.

Reglas para sílabas que comienzan con consonante de ataque

1. Una consonante entre dos vocales**: V_1 C V_2**
   En este caso la consonante puede ser ll y rr y la segunda sílaba comienza en **C**. Ej: u-no, a-mi-go, a-lla-na-do.
2. Dos consonantes contiguas rodeadas de vocales**: V_1 C_1 C_2 V_2**
   a) Cuando **C_1** sea **p**, **c**, **b**, **g** o **f** y **C_2** sea **r** o **l** la segunda sílaba empieza en **C_1**. Por ejemplo: a-bra-zo, i-glú, o-pri-me, o-clu-ir, o-fre-ce.
   b) Cuando **C_1** sea **d** o **t** y **C_2** sea **r** la segunda sílaba empieza en **C_1**. Ej: cua-tro, ce-dro.
   c) Con cualquier otra combinación de dos consonantes entre vocales la segunda sílaba empieza en **C_2**. Por ejemplo: is-la.
3. Tres consonantes contiguas entre vocales**: V_1 C_1 C_2 C_3 V_2**
   a) Cuando las consonantes **C_2 C_3** sigan la regla 2.a o la 2.b la segunda sílaba empieza en **C_2**. Ej: des-gra-cia.
   b) Cuando **C_1** sea **b**, **d**, **n**, **m**, **l** o **r** y **C_2** sea **s** la segunda sílaba empieza en **C_3**. Ej: abs-te-mio, ins-tar, háms-ter, sols-ti-cio, pers-pi-caz.
   c) Cuando **C_1** sea **s** y **C_2** sea **t** la segunda sílaba empieza en **C_3**. Ej: post-da-ta, ist-mo.
4. Cuatro consonantes contiguas entre vocales: **V_1 C_1 C_2 C_3 C_4 V_2**
   Sólo es posible que las dos primeras consonantes se comporten como en 3.b o 3.c y las dos últimas como en 2.a. La segunda sílaba siempre empieza en **C_3**. Por ejemplo: post-pro-duc-ción, trans-plan-te, ads-cri-bir.

Reglas para sílabas que no tienen ataque

5. Dos vocales consecutivas o separadas por una h: **V_1 V_2** o **V_1** h **V_2**
   a) Cuando forman diptongo las vocales no se separan.
      I. Forman diptongo una vocal abierta (**a,e,o**) con o sin tilde, seguida de una cerrada (**i,u**), sin tilde. Ej: a-plau-so.
      II. Forman diptongo una vocal cerrada (**i,u**) sin tilde seguida de una abierta (**a,e,o**) con o sin tilde. Ej: cua-dro, ciu-dad, fun-ción
      III. Forman diptongo dos vocales cerradas distintas, es decir **iu** o **ui**. Una de las dos puede llevar tilde y la **u** puede llevar diéresis (**ü**). Ej: cui-dar, a-cuí-fe-ro, a-güe-ro.
   b) Cuando forman hiato las dos vocales sí se separan por lo que la segunda sílaba comienza en **V_2**.
      I. Forman hiato la combinación de una vocal abierta sin tilde y una vocal cerrada con tilde en cualquier orden. Ej: ra-íz, Ma-rí-a.
      II. Forman hiato dos vocales iguales, una de las cuales puede llevar tilde: Ej: po-se-er, chi-i-ta, chi-í.
      III. Forman hiato dos vocales abiertas distintas, una de las cuales puede llevar tilde. Ej: ca-er, o-cé-a-no.
   c) La **h** intercalada entre dos vocales no influye en la consideración como diptongo o hiato de una secuencia de dos vocales; en caso de hiato, la **h** siempre se une a la segunda vocal por lo que, en ese caso, el comienzo de la segunda sílaba está en la **h**. Ej: pro-hí-be, prohi-bir.
6. Tres vocales consecutivas: **V_1 V_2 V_3**
   Cuando **V_1** y **V_3** son vocales cerradas sin tilde y **V_2** es abierta, pudiendo llevar tilde, se trata de un Triptongo y no hay separación silábica. Es decir, la segunda sílaba comienza después de **V_3**. Ej: a-pre-ciáis, buey)

Ejemplo de aplicación de las reglas

La primera búsqueda de una separación silábica debe comenzar desde el comienzo de la palabra. Si se produce la coincidencia con alguna de las reglas numeradas del 1 al 5, la siguiente búsqueda debe comenzar desde la vocal V_2 (incluyendo a ésta) ya que las reglas buscan patrones desde la última vocal del núcleo de la sílaba actual, con el fin de encontrar el ataque de la siguiente. Por ejemplo, supongamos que la palabra a tratar es *abstemio*. Consideraremos que los caracteres empiezan a numerarse en 0. La posición de corte indica el primer carácter de la siguiente sílaba. La secuencia de búsquedas y coincidencias es:

Posición de inicio	Regla verificada	Texto reconocido	Posición de corte
0	3.b. V_1 C_1 C_2 C_3 V_2	abste	3 (abs-te)
4	1. V_1 C V_2	emi	5 (e-mi)
6	5.a. V_1 V_2	io	-

Si la regla que se verifica es la 6, la siguiente búsqueda empezará desde V_3.

Anexo 2. Reglas para detectar la sílaba y la vocal tónica

Como sabemos**, si una palabra tiene tilde** (sobre una vocal, obviamente) la sílaba en la que se encuentra dicha vocal es la sílaba tónica de la palabra y la vocal tónica, evidentemente, es la que lleva la tilde. Sin embargo, también es sabido que la mayor parte de las palabras en español no tienen tilde y no por ello dejan de contener sílaba y vocal tónicas.

Para las palabras que no tienen tilde aplicaremos las siguientes reglas:

- Si una palabra sin tilde termina en vocal, o en las letras 's' o 'n', tiene dos o más sílabas, la penúltima sílaba es la tónica. Por ejemplo: Examen
- Si una palabra sin tilde termina con una consonante distinta de 's' o 'n', la última sílaba es la tónica. Por ejemplo: ababol

Para determinar la vocal tónica dentro de una sílaba tónica con varias vocales actuaremos así:

- En los diptongos la vocal tónica es la abierta o la segunda cerrada.
- En los triptongos la vocal tónica es la central.

Evidentemente, estas reglas son una versión simplificada y hay aspectos del castellano que no se contemplan y otros como, por ejemplo, la acentuación de adverbios terminados en mente que no se están tratando.

Finalmente, podemos clasificar las palabras en agudas, llanas, esdrújulas según si su sílaba tónica es la última, penúltima, antepenúltima, respectivamente. Si la tónica es cualquier sílaba anterior a la antepenúltima, se dice que es o sobresdrújula.

Anexo 3. Sintaxis extendida de ER

La manera de escribir expresiones regulares varía de unas aplicaciones a otras, aunque hay un núcleo común en las distintas variantes de sintaxis de las expresiones regulares (*regex flavors*). En este anexo se introduce la sintaxis más común usada en lenguajes de programación como Python que es una extensión de la sintaxis teórica que se estudia en teoría. La sintaxis que se usa en algunos programas como *egrep, vi* o *flex* que forman parte de la distribución de Linux es algo diferente.

Cualquier aplicación que permite usa expresiones regulares para realizar operaciones de *regular pattern matching*, como validación de formato de cadenas, búsqueda o sustitución usando un patrón dado por una ER, cuenta con un *Motor de ER (regex engine). Este* se encarga de compilar o traducir la ER en una implementación de autómata finito cuya tabla de transición se usa en los métodos para procesar las cadenas. Dicha implementación es transparente al usuario/programador, que se ocupa sólo de escribir la ER y de aplicar los métodos que proporciona el *regex engine* para resolver los problemas de procesamiento de cadenas de patrón regular. Lo más importante es tener en cuenta que el proceso de compilación es mucho más costoso en tiempo que el uso de la ER ya compilada.

Funcionamiento básico de EditPad Pro

EditPad Pro es un editor de textos que cuenta con un regex engine. Puede descargarse de:

http://www.editpadpro.com/download.html (free trial)

En recursos del Aula Virtual hay ficheros que pueden usarse para probar los ejemplos de ER que estudiaremos a continuación. Por ejemplo:

- Abre el fichero *prueba-EditPad-varios.html,*
- Selecciona en el menú *Search* la opción *Show Search Panel.*
- Selecciona en el panel *Search&Replace* las casillas *Regular Expression y Case Sensitive.*
- Con *Find First* busca en el fichero cargado **la primera cadena del texto S coincidente con la expresión regular R** introducida en *Search* (primera cadena S tal que *R casa con S*) y la resalta en gris.
- Con *Next* busca la siguiente cadena coincidente **desde la posición en que finalizó el *matching* anterior**. Si en el texto no hay ninguna cadena que se ajuste al patrón de R, el editor lo advierte con una cruz roja.
- En la barra de *Search&Replace* se puede activar *Highlight*. Esto hace que se resalten **todas las cadenas coincidentes**, como si se hiciera *Next* de forma repetida. El resalte en algunos casos puede resultar confuso si se solapan las coincidencias.

Precaución con espacios en blanco: en el panel *Search&Replace* hay que tener cuidado de no poner espacios en blanco y otros caracteres que no sean parte del patrón de la ER, sobre todo al hacer *copy-paste* de Word a EditPad. Los blancos los señala con un punto gris claro para evitar confusión.

Tabla de sintaxis extendida

A continuación, introducimos un resumen de la sintaxis extendida de ER.

Expresiones regulares que describen un único carácter o un conjunto de caracteres		
Sintaxis	Descripción	Explicación y ejemplos de emparejamiento (Expresión regular en azul y cadena literal en amarillo)
c	c representa un carácter literal, que es cualquier carácter que no sea uno de los siguientes caracteres especiales para la sintaxis [ ] \ ^ $. | ? * + ( ) { }	a casa con a Una ER que consiste en una secuencia de caracteres literales casa exactamente con la cadena que forma la ER: hola casa con hola (y ninguna más en ***modo case sensitive***, que diferencia mayúsculas de minúsculas) hola no casa con holaMundo ni con HoLA
****e	e representa un carácter especial (listados arriba) y al poner \e, la barra de escape hace que el carácter especial e pierda su significado y entonces \e describe al carácter literal e.	8\+2 casa con 8+2 (y ninguna más) 8+2 no casa con 8+2
\n \r \t	ERs para describir caracteres de LF (*line feed*), CR (*carrige return*) y tabuladores, respectivamente.	La ER \r\n describe a un *newline* en Windows y \n a un *newline* en Unix/Linux
[secCar]	[secCar] describe un conjunto de caracteres literales. La ER casa con cualquier carácter individual que aparezca en la secuencia de caracteres secCar. Los caracteres especiales pierden su significado dentro de los corchetes, pasan a ser literales y no hay que ponerles la barra de escape.	[hola] casa con h (también los caracteres o, l, y a y ninguno más) [hola] no casa con hola [hH][oO][Ll][aA] casa con hola y también con HoLa. Es decir sólo con la palabra "hola" con letras mayúsculas o minúsculas. [+*] casa únicamente con + o con *
[^secCar]	Al poner ^ (gorro o caret) se complementa el conjunto de caracteres que va detrás con respecto al conjunto total de caracteres.	[^+*] casa con cualquier carácter que NO sea + ni *
[**c1-c2**]	[c1-c2] describe un conjunto de caracteres dado por un rango y casa con cualquier carácter individual cuyo código ASCII esté comprendido entre el código del carácter c1 y el código del carácter c2. Pueden aparecer varios rangos dentro de los corchetes y se puede complementar el rango con ^	[0-9] equivale a la ER (0|1|2|3|4|5|6|7|8|9) que casa con cualquier dígito decimal [^0-9] casa con cualquier carácter que NO sea un dígito [a-z] casa con cualquier letra minúscula ASCII [a-zA-Z] es una ER con dos rangos y casa con una letra mayúscula o minúscula ASCII [a-z][a-z][0-9] casa pe2, zb7,… y no casa con e2 ni con variable1

\d	La ER \d es una abreviatura (*sorthand*) que describe al conjunto de los dígitos decimales.	\d equivale a [0-9] \d\d\d casa con 885 y no casa con 12
\w	La ER \w es una abreviatura que describe al conjunto de los *caracteres tipo word (palabra):* son los dígitos decimales, letras ASCII y _ (subrayado).	\w equivale a [0-9a-zA-Z_] \w[\d.,] casa con h1 con r, y con 9. (las abreviaturas pueden usarse dentro de corchetes)
\p{L} **\p{Lu}** **\p{Ll}**	\p{L} describe al conjunto de letras Unicode (mayúsculas o minúsculas). Permite que las letras acentos, diéresis, signos fonéticos, etc. \p{Lu} sólo para letras Unicode mayúsculas \p{Ll} sólo para letras Unicode minúsculas	La ER [a-zA-Z]+ no casa con ambigüedad ni con España ni con fácil. \p{L}+ casa con cualquiera de las anteriores. \p{Ll}+ casa con fácil y no casa con España \p{Lu}+ casa con ESPAÑA y no casa con España
\s	La ER \s es una abreviatura que describe el conjunto de *caracteres de espaciado* (blanco, \t, \r o \n). Equivale a [\n\r\t]	hola[\d\s]mundo casa con hola1mundo con hola mundo y no casa con holamundo
.	El punto (dot) es una ER que casa con cualquier carácter *excepto* los caracteres de nueva línea \r o \n	car. casa con cara caro car1 car. car+ car\. casa con car. (y ninguna cadena más)
Expresiones regulares con operador de alternancia y paréntesis de agrupación (*alternation, grouping*)		
R1|R2	El | (pipe) es el operador de unión o alternancia de ER. La expresión **R1|**R2 tiene coincidencia con cadenas que se ajustan a R1 o a R2. Es el operador de menor precedencia y con los paréntesis se agrupa una subexpresión para alterar la precedencia.	America|Europe|Canada casa con America con Europe con Canada y con ninguna cadena más. (hola|bye)Mundo casa con holaMundo con byeMundo y con ninguna más. (hola|bye)Mundo equivale, por la propiedad distributiva, a holaMundo|byeMundo pero es mejor la primera forma.
Expresiones regulares con operadores de opcionalidad y repetición (*quantifiers*)		
R*	La estrella * es el operador de clausura. R* indica cero o más repeticiones de cadenas que casan con el patrón R	[a-z]\d* casa con a1234 con b2 con c y no casa con 12 ni con d-83
R*?	*? es el operador de clausura perezosa (*lazy star*). Cuando se usa R*? en una búsqueda, el *regex engine* selecciona la cadena más corta que se ajusta al patrón R*.	Buscando con <.*> en una línea que contiene <h3>CHARACTER SET</h3> y sin más ángulos > en línea, se selecciona <h3>CHARACTER SET</h3> Buscando con <.*?> selecciona desde <h3>. Es decir, sólo hasta el primer ángulo de cierre.

R+	Se corresponde con el operador de clausura positiva de lenguajes. R+ indica una o más repeticiones del patrón R. Equivale a RR*	[a-z]\d+ casa con a1234 con b2 y no casa con c ni con 12 ni con d-83
R+?	+? es versión perezosa del operador + (*lazy plus*). Análogo a R*? pero obligando al menos una coincidencia con R	Buscando con <.+?> en una línea que contiene <> </br> y sin más > en la línea, se selecciona <> </br> Si se busca con <.*?> entonces se selecciona <>
R?	El operador ? (no precedido de * ni +) es un operador de opcionalidad e indica que R es opcional. R? equivale a (R|λ)	abc? casa con ab y con abc y con ninguna cadena más a(bc)? casa con a y con abc y con ninguna cadena más
R{n} R{n,m} R{n,}	Indica n repeticiones del patrón R Indica entre n y m repeticiones de R Indica n o más repeticiones de R	a{3} casa con aaa y con ninguna cadena más v\d{1,3} casa con v5 y con v60 y con v623 y no casa con v ni con v8965 [A-Z]{3}\d{2,} casa con BXZ47 y con ABC123 y con RSA1234 y no casa con B43 ni con 576
ER con anclas y delimitadores de palabras *(anchors, word boundaries)*		
^R	EL ^ cuando va fuera de corchetes tiene coincidencia con la posición de comienzo de una cadena o línea de texto. Cuando se usa ^R hay matching con una cadena S que se ajusta al patrón R y aparece como prefijo.	Buscando con ^<[a-zA-Z]+> se selecciona en una línea que comience por Si antes del carácter '<' hay espacios u otro carácter entonces no hay *matching* en esa línea.
R$	Análogo al anterior pero con la posición de final de cadena o línea de texto.	Buscando con <[a-zA-Z]+>$ se selecciona <head> en una línea que acaba por <head> Buscando con la ER ^\d+$ selecciona cadenas de uno o más dígitos que ocupan toda la línea.
\bR\b	\b es un delimitador de palabra y en una búsqueda con \bR\b se selecciona una cadena que casa con R y está delimitada a la izquierda y a la derecha por caracteres que NO son letras o dígitos.	Buscando con \b\d+\b se selecciona 8859 en una línea que contiene charset=ISO-8859- Si fuera charset=ISO8859- no habría matching con 8859 Se usa el ancla \b para emparejamiento con palabras completas.
ER con operadores de contexto (*lookahead y lookbehind*)		
R1(?=R2)	*Positive lookahead.* Busca cadenas que verifican R1 siempre y cuando vayan seguidas de cadenas que verifican R2. La cadena coincidente con la expresión regular completa es sólo la que verifica R1.	[a-z]+(?=@) selecciona alf@um.es

R1(?!R2)	*Negative Lookahead*. Busca cadenas que verifican R1 siempre y cuando no vayan seguidas de cadenas que verifican R2. La cadena coincidente con la expresión regular completa es sólo la que verifica R1.	[a-z]+(?![a-z@]) selecciona alf@um.es
(?<=R2)R1	*Positive Lookbehind*. Busca cadenas que verifican R1 siempre y cuando vayan precedidas de cadenas que verifican R2. La cadena coincidente con la expresión regular completa es sólo la que verifica R1.	(?<=\.)[a-z]+ selecciona alf@um.es
(?<!R2)R1	*Negative Lookbehind*. Busca cadenas que verifican R1 siempre y cuando no vayan precedidas de cadenas que verifican R2. La cadena coincidente con la expresión regular completa es sólo la que verifica R1.	(?<![.a-z])[a-z]+ selecciona alf@um.es

Modificadores de expresiones regulares

Los modificadores de las expresiones regulares sirven para alterar el modo en que el regex engine de cierta herramienta realiza las operaciones de *pattern matching* a partir de una expresión regular y de ese modo se puede acortar la expresión regular. Sólo deben usarse cuando realmente tengan algún efecto, no de forma indiscriminada, pues disminuyen la eficiencia.

Modificador para modo "dot matches newline"

En métodos con ER en Python y en editores de texto como EditPad Pro (ver sección de funcionamiento de EditPad más adelante), el punto (*dot*) no tiene coincidencia por defecto con los caracteres de nueva línea (como \r o \n).

Ejemplo: si se usa la ER </head>.*<BODY> para buscar en un texto donde aparecen las etiquetas html </head> y luego <BODY> separadas por newlines entonces el *regex engine* no encuentra ninguna cadena coincidente.

Para que el punto tenga coincidencia con caracteres de nueva línea (modo *dot matches newline)*, sin que dependa de la herramienta concreta, se usa el *modificador de dot* **(?s)** delante de laER.

Ejemplo: con (?s)</head>.*<BODY> se selecciona desde </head> hasta <BODY> incluido, aunque esas etiquetas html vayan en líneas separadas.

Modificador para modo "case insensitive"

El modo por defecto en los métodos con ER en Python es *case sensitive*, esto es, se distingue entre mayúsculas y minúsculas en las letras en las operaciones de matching. Para activar el modo contrario *case insensitive* para una ER sin que dependa de la herramienta concreta se pone el *modificador de case* **(?i)** delante de laER.

Ejemplo: (?i)hola casa con HoLA hola HOLA (no se distinguen mayúsculas de minúsculas)

Mezcla de modificadores

Los modificadores pueden juntarse, se puede poner **(?is)R** para indicar modo *insensitive* y *dot matches newlines* para la expresión regular R que va detrás de los modificadores.

Ejemplo: Con (?is)</head>.*<body> se selecciona desde la etiqueta </head> hasta <body> incluido, aunque esas etiquetas vayan en líneas separadas y con independencia de que las palabras de etiqueta "head" y "body" vayan con las letras en mayúsculas o minúsculas. Tiene sentido poner el modificador de case porque en el lenguaje Html no se diferencian las mayúsculas de las minúsculas en las etiquetas. La búsqueda también tendría éxito si, por ejemplo, la palabra clave "head" aparece en el documento escrita como "Head" o "HEAD".

Capturas y referencias a capturas (backreferences)

Cada una de las subexpresiones **entre paréntesis** que aparecen dentro de una expresión regular forma un **grupo**. Los grupos se enumeran desde 1 contando los paréntesis de apertura en una ER de izquierda a derecha y el grupo 0 siempre es la propia expresión regular, aunque no vaya entre paréntesis.

Ejemplo: En la ER ([A-Z]+)(\d+)([a-z]+) aparecen paréntesis que no son necesarios para alterar la precedencia, pero al ponerlos se distinguen 4 grupos en la expresión regular:

El **grupo 0** siempre es la propia expresión regular, en este caso ([A-Z]+)(\d+)([a-z]+)

El **grupo 1** consiste en la subexpresión regular: ([A-Z]+)

El **grupo 2** es: (\d+)

El **grupo 3** es: ([a-z]+)

Los grupos de una ER pueden ser usados para que el *regex engine* haga una captura de las subcadenas que se ajustan al patrón de cada grupo, proceso al que nos referimos de forma abreviada como **captura de grupos**. Las capturas **pueden referenciarse** mediante la referencia **\i** donde i es el número del grupo, por ej. \1 (captura de grupo 1), \3 (captura de grupo 3), o bien \0 para la captura de la cadena que se ajusta a la ER completa.

Ejemplo: Usando la ER ([A-Z]+)(\d+)([a-z]+) con la cadena AULA12b:

El **grupo 0** AULA12b

El **grupo 1** AULA

El **grupo 2** es: 12

El **grupo 3** es: b

¿Cómo se usan las referencias a las capturas para hacer sustituciones con ER?

Una de las aplicaciones del proceso de captura de grupos de una ER es para realizar una sustitución que no es por cadena fija (la típica), sino que se sustituye por una cadena que depende del patrón de la ER.

Ejemplo: Supongamos que queremos buscar una cadena que se ajusta al patrón ([A-Z]+)(\d+)([a-z]+) para sustituirla por la cadena que consiste en las letras minúsculas de la cadena original seguidas de un guion, luego los dígitos encerrados entre corchetes, seguido de un guion y finalmente las letras mayúsculas. Por ejemplo para sustituir AB12xy por xy-[12]-AB.

Obviamente esto no se puede hacer con un *replace* o sustitución simple como se hace con los procesadores de texto de uso general. En el editor EditPad, que es un editor profesional para programadores y permite usar ER, se pondría lo siguiente en el panel *search&replace*:

SEARCH: ([A-Z]+)(\d+)([a-z]+) REPLACE: \3-[\2]-\1

Con *next+replace* sucesivas veces podemos sustituir cada cadena coincidente con la ER del campo *Search*. Con *Replace All* se sustituyen de golpe todas las cadenas coincidentes.

Es importante tener en cuenta que la cadena sustituida es literal excepto por las referencias a los grupos capturados.

También es posible nombrar los grupos y utilizar los nombres en lugar de sus posiciones a la hora de indicar los reemplazos. Para nombrarlo se incluye ?P seguido del identificador entre los símbolos < y > al principio del grupo. Después, podemos referenciar los grupos capturados con \g<id> siendo id el identificador del grupo.

Ejemplo: Si vamos buscando la planta y la letra de una vivienda en una dirección con la ER (\d+)([a..zA..Z]) podríamos darle nombre a cada uno de los dos grupos así: (?P<planta>\d+)(?P<letra>[a..zA..Z]). De este modo, la sustitución podría hacerse así:

SEARCH: (?P<planta>\d+)(?P<letra>[a..zA..Z]) REPLACE: Piso \g<planta> letra \g<letra>

Finalmente, cuando se quieren usar paréntesis para agrupar partes de ER pero no se quiere que dichos grupos puedan ser referenciados, basta con añadir la secuencia :? a continuación del primer paréntesis.

Ejemplo: Buscando con la ER ([a..z]+)(:?\d+)([a..z]+) los grupos posibles a usar, además del 0, serían el 1 y el 2 que coincidirían con las cadenas de 1 o más letras minúsculas. Pero los dígitos que las separan no podrían referenciarse.

Uso de referencias dentro de una ER para búsquedas más complejas

Las referencias permiten construir **expresiones regulares aumentadas más potentes** (describen lenguajes que no son regulares). Este tipo de expresiones regulares aumentadas no es traducible a un autómata finito puro, porque se requiere guardar en memoria la captura (parte de la cadena que se procesa) y el modelo de autómata finito es el de una máquina sin memoria.

Ejemplo: Supongamos que queremos buscar en un texto dos palabras repetidas, entendiendo que son cadenas iguales formadas por letras Unicode mayúsculas o minúsculas y las palabras van separadas por uno o más caracteres de tipo espacio (\s).

Por ejemplo, "HOLA Pedro Hola Juan" debería seleccionase todo el texto entre los dos saludos. Y da lo mismo que vayan en la misma línea o en líneas separadas y que una de las palabras lleve minúsculas. También requerimos que se seleccionen **palabras completas**, no fragmentos de palabras. Por ejemplo, si en una línea aparece "es un uno" entonces no se debería resaltar "Es un uno" porque "un" forma parte de una palabra, no es una palabra completa. Esto indica que debemos usar el delimitador de palabra \b. En EditPad pondríamos la siguiente ER para la búsqueda

SEARCH: \b(\p{L}+)\b\s+\b\1\b

Anexo 4. Cálculo de Rimas

A la hora de escribir un poema o los versos de un tema musical es habitual incluir rimas para aportar musicalidad. La rima se produce en la terminación de las palabras finales de dos o más versos. En español hay dos tipos de rima: tipo asonante y tipo consonante. Siguiendo un enfoque simplificado, aquí consideraremos que dos palabras **riman** con rima **asonante** si ambas tienen la misma vocal tónica y coinciden en el resto de vocales hasta el final. Por otra parte, consideraremos que dos palabras **riman** con rima **consonante** si tienen la misma vocal tónica y coinciden en vocales y consonantes hasta el final.

A continuación se muestran varios ejemplos de palabras y otras que riman con ellas (marcamos las terminaciones en amarillo):

- Melocotón
  - tiene rima asonante con: dos, feróz , girasol y calentador
  - tiene rima consonante con: son, camión, tiburón y conversación
- Fresa
  - tiene rima asonante con: teca, regresan, regadera
  - tiene rima consonante con mesa, calesa y frambuesa
- Llámalo
  - tiene rima asonante con: rebájalo, cántaros y relámpagos
  - tiene rima consonante con ámalo, trámalo y derrámalo

EJERCICIO OPCIONAL: Para conseguir 1,5 puntos extra en la práctica debes añadir a la aplicación la capacidad de calcular qué terminación de rima asonante y qué terminación de rima consonante tiene cierta palabra y ofrecer al usuario la posibilidad de encontrar palabras que rimen con una dada. Para ello, puedes crear un diccionario de terminaciones en el que las claves serán las distintas terminaciones que se vayan encontrando al analizar palabras y como valor tendremos la lista de palabras que rimen igual, de entre las que ya se hayan analizado anteriormente. Lo más conveniente será que tengas un diccionario para rimas asonantes y otro para rimas consonantes y que añadas, a la información guardada por cada palabra del diccionario original de palabras, sus dos tipos de rimas. Finalmente, para representar las terminaciones de rimas asonantes puedes usar una simple cadena.

Anexo 5. Justificación de texto

La justificación de un texto consiste en alinearlo a izquierda y derecha de este de modo que la longitud de los renglones de cada párrafo sea la misma en todos e igual a la deseada. La justificación se puede hacer asumiendo que todas las letras tienen la misma anchura o que cada una tiene un ancho diferente. Además, se debe tener en cuenta la anchura del espacio mínimo entre palabras, lo que llamamos el espacio en blanco mínimo.

Para justificar un párrafo basta con ir añadiendo palabras y espacios en blanco mínimos entre ellas hasta que nos pasemos de la anchura máxima establecida para el justificado. En ese momento, una estrategia simple, aunque no perfecta, consiste en dividir en sílabas la última palabra de modo que, tras añadir el guion que indique el corte de la palabra, no nos pasemos de la anchura máxima. Como es posible que en este momento aún nos quede espacio libre, se puede añadir espacio entre las palabras del renglón actual. En el caso de estar usando una fuente con anchura de letra fija añadiremos espacios en blanco mínimos completos.

Evidentemente, el siguiente renglón comenzará con el resto de la palabra que haya quedado cortada en el anterior.

EJERCICIO OPCIONAL: Para conseguir 1,5 puntos extra en la práctica puedes añadir una opción al menú principal de la aplicación que permita seleccionar un fichero de texto y una anchura de párrafo. A continuación, la aplicación mostrará el texto justificado. Opcionalmente, se podría pedir al usuario que indicara un fichero donde escribir el texto justificado.