Teoría de Lenguajes

Segundo cuatrimestre de 2023

Trabajo práctico: *tlengrep*

Primera parte

Fecha de entrega: jueves 12 de octubre de 2023 a las 23:59

Versión del enunciado: 1 (4 de septiembre de 2023)

Las **expresiones regulares** son ampliamente utilizadas porque permiten describir patrones de texto de una forma concisa, sencilla y legible. El objetivo de este trabajo práctico es implementar una herramienta sencilla que nos permita encontrar las ocurrencias de una expresión regular en un texto, de una forma similar a la aplicación `grep`.

El trabajo práctico está dividido en dos etapas:

- En la **primera parte** nos concentraremos el "motor" de la herramienta. Utilizando una representación interna de expresiones regulares como objetos, implementaremos la funcionalidad de buscar las ocurrencias de las mismas en un texto. Para esto deberemos poner en práctica nuestros conocimientos sobre lenguajes regulares y autómatas finitos.
- En la **segunda parte**, construiremos un *parser*, es decir, un programa que recibirá expresiones regulares escritas como texto y las convertirá a la representación interna que utiliza nuestro motor. Luego, unificaremos ambas partes para obtener una herramienta completa.

1. Descripción del problema

Se desea construir un programa que reciba como parámetro una expresión regular y un archivo de texto. El programa deberá procesar línea por línea el archivo, y filtrar aquellas líneas que coincidan con la expresión regular.

1.1. Características del programa

La solución del trabajo práctico será un programa escrito en Python 3 [1]. La forma de invocación del programa será la siguiente:

```
$ python3 tlengrep.py <expresión regular> <archivo de texto>
```

donde `<expresión regular>` es la expresión regular que se desea buscar, y `<archivo de texto>` es el archivo de texto en el que se desea buscar.

En el programa terminado, `<expresión regular>` será una cadena de texto representando la expresión regular. Por el momento, proporcionaremos la ruta hacia un módulo de Python donde una variable llamada `__regex__` contendrá su definición en nuestra representación interna. El programa contará con una opción adicional, `-m`, que habilitará este modo de funcionamiento. Por ejemplo,

```
$ python3 tlengrep.py -m tests.regexes.r00 <archivo de texto>
```

obtendrá la expresión regular desde el archivo `tests/regexes/r00.py`.

El argumento `<archivo de texto>` será opcional, y en caso de no especificarse, el programa deberá leer desde la entrada estándar.

El programa deberá imprimir por la salida estándar exactamente aquellas líneas del archivo de entrada que coincidan (en forma completa, de principio a fin) con la expresión regular, sin modificarlas.

1.2. Expresiones regulares

Para esta primera entrega, usaremos la siguiente definición recursiva de expresiones regulares:

- $\varnothing$ es una expresión regular que denota el lenguaje vacío ($\varnothing$).
- λ es una expresión regular que denota el lenguaje que contiene solo la palabra vacía (Λ).
- Dado un carácter a, a es una expresión regular que denota el lenguaje $\{a\}$.
- Si E_1 y E_2 son expresiones regulares, entonces:
  - E_1E_2 es una expresión regular que denota el lenguaje $\mathcal{L}(E_1)\,\mathcal{L}(E_2)$.
  - $E_1|E_2$ es una expresión regular que denota el lenguaje $\mathcal{L}(E_1) \cup \mathcal{L}(E_2)$.
  - E_1^* es una expresión regular que denota el lenguaje $\mathcal{L}(E_1)^*$.
  - E_1^+ es una expresión regular que denota el lenguaje $\mathcal{L}(E_1)^+$.

La representación interna de las expresiones regulares será a través de una clase abstracta, `RegEx`, y una subclase concreta para cada uno de los casos recién mencionados: `Empty`, `Lambda`, `Char`, `Concat`, `Union`, `Star` y `Plus`, respectivamente.

1.3. Funcionalidad de búsqueda

Para implementar la búsqueda de apariciones en el texto de manera eficiente, deberán utilizar el **autómata finito determinístico** (AFD) de **estados mínimos** correspondiente a la expresión regular. La conversión se realizará utilizando el método de Thompson (visto en la teórica), que pueden encontrar en [2, Sección 3.2.3].

Dado que la transformación de la expresión regular a un AFD es un proceso costoso, es importante que no repitan este cómputo para cada línea del archivo de entrada.

2. Código base

Para facilitar la implementación, junto al presente enunciado, la cátedra provee un proyecto inicial a partir del cual desarrollar la solución. Está permitido hacer todas las modificaciones que consideren necesarias en el código base, siempre y cuando la solución respete los requisitos expresados en este enunciado.

2.1. Estructura

El código base está estructurado como se muestra en la Figura 1, e incluye:

- Un **ejecutable principal** que maneja la toma de parámetros, la entrada y la salida.

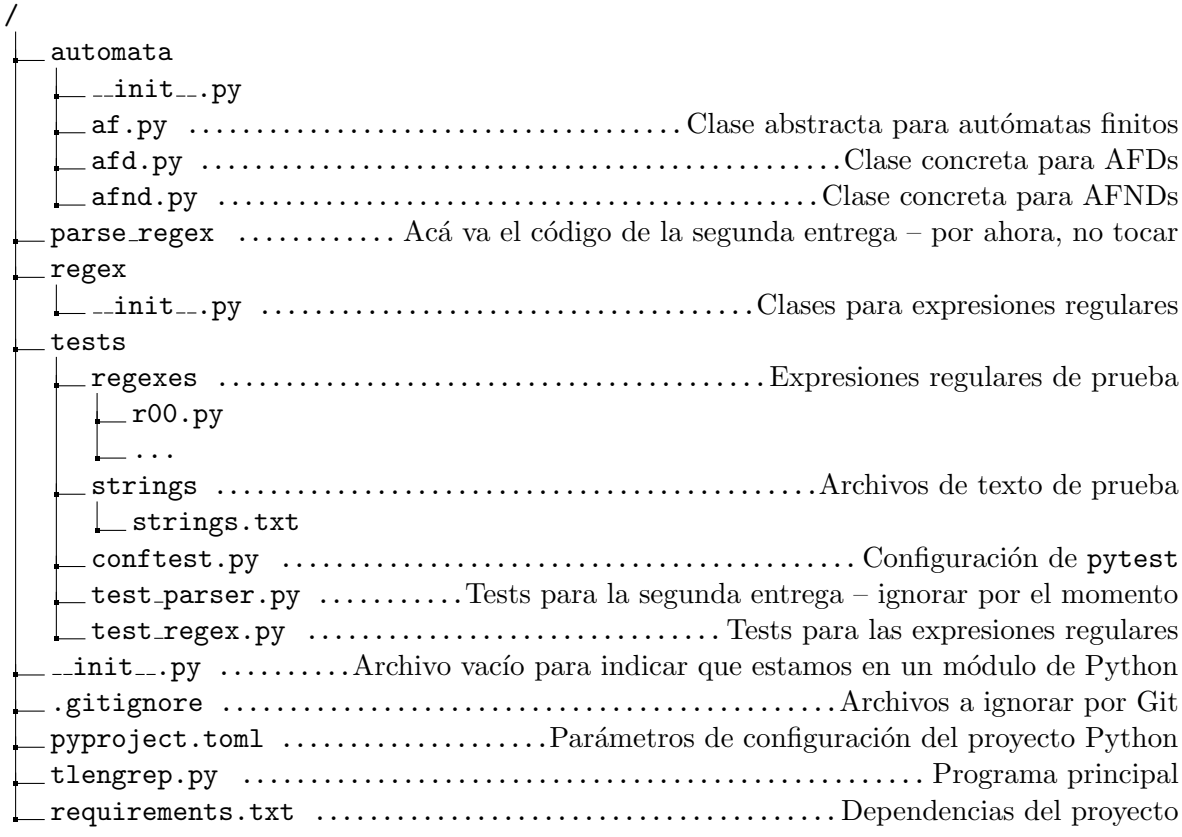

Figura 1: Estructura del proyecto

- Un módulo para la representación de **expresiones regulares**, con una clase abstracta y una subclase concreta para cada uno de los casos de la definición. Se incluye funcionalidad para imprimir las expresiones regulares por pantalla y una implementación *naive*, recursiva, de la evaluación de una cadena de entrada, que no es eficiente.

  Recomendamos no modificar la interfaz de esta clase (salvo para agregar nueva funcionalidad) para evitar problemas en la segunda entrega.

- Un módulo para la representación de **autómatas finitos**, tanto determinísticos como no determinísticos. Se incluyen funciones auxiliares básicas, por ejemplo, para imprimir una tabla con la función de transición.

- **Casos de prueba**. Es condición necesaria (pero no suficiente) para la aprobación que su solución pase exitosamente todos los casos de prueba provistos. Los mismos están implementados usando la biblioteca `pytest` [3], y pueden ejecutarse con el comando

  ```
  $ pytest -k test_regex.py
  ```

  Recomendamos fuertemente no contentarse con estos casos y agregar todos los que consideren necesarios. Para hacerlo pueden agregar nuevos métodos a las clases definidas dentro de la carpeta `tests`, o bien crear nuevos archivos de prueba. Tanto los nombres de los archivos como de los métodos que definan deben tener el prefijo `test_` para ser detectados por `pytest` (ver la documentación para más detalles). Cualquier método de test que tenga algún parámetro llamado `strings` será ejecutado una vez por cada archivo de texto que exista en la carpeta `tests/strings`, y recibirá como argumento una lista con las líneas del archivo.

2.2. Consideraciones

- El código provisto fue probado con Python 3.9.16. Recomendamos utilizar la misma versión para evitar problemas de compatibilidad. En caso de no ser posible, y de encontrar algún problema, sugerimos reportarlo a la cátedra.
- El código tiene dos dependencias (detalladas en el archivo `requirements.txt`): `pytest` y `tabulate`. Para instalarlas, ejecutar `pip3 install -r requirements.txt`. Recomendamos usar un entorno virtual [4] para instalar las dependencias del proyecto.
- El código contiene *type hints* (anotaciones de tipos) a modo de documentación. Python ignora estas anotaciones al momento de la ejecución, aunque son útiles a la hora de leer el código. No es un requisito que su solución contenga estas anotaciones.

3. Condiciones de entrega

- El trabajo práctico es grupal y debe realizarse en grupos de **tres integrantes**.
- La fecha límite de entrega es el jueves 12 de octubre de 2023 a las 23:59, **sin excepciones**.
- La entrega se realizará a través de **GitHub Classroom** (ver el apartado 3.3 para más detalles). Para la corrección se considerará la última versión del código que se haya publicado (*pusheado*) en la rama `main` antes de la fecha límite de entrega. Cualquier *commit* enviado con posterioridad será ignorado.
- Recomendamos fuertemente no utilizar herramientas de inteligencia artificial (como GitHub Copilot, ChatGPT, etc.) para asistir en la realización de la solución. El uso de este tipo de herramientas va en contra del objetivo del trabajo práctico, que es que desarrollen sus propias habilidades.

3.1. Código fuente

La entrega deberá incluir un directorio llamado `tlengrep` que contenga el **código fuente** de un programa que implemente la funcionalidad descripta en las secciones anteriores. Es importante que:

- Incluyan **instrucciones claras** para ejecutar el programa (sugerimos agregar un archivo `README.md` o incorporarlas en el informe).
- El código sea prolijo, claro y ordenado, siguiendo **buenas prácticas** de programación.
- Agreguen todos los **casos de test** que consideren necesarios.
- No utilicen bibliotecas externas para implementar partes relevantes de la solución del problema.

3.2. Informe

La entrega deberá incluir un directorio llamado `informe` que contenga un **informe** de carácter **breve** donde expliquen la solución adoptada y cualquier decisión que hayan tomado y consideren relevante documentar.

Además, deberán comparar la **eficiencia** de su solución, en términos de tiempo de ejecución, con la implementación *naive* provista por la cátedra. Se espera que elijan al menos una expresión regular de complejidad no trivial y que muestren qué sucede para cadenas de distinta longitud

y para distintas cantidades de cadenas. El informe deberá contener los resultados obtenidos y un breve análisis de los mismos, explicando las conclusiones que pueden extraer de ellos.

3.3. Uso de GitHub Classroom

Utilizaremos la plataforma **GitHub Classroom** tanto para facilitarles el código base del trabajo práctico como para recibir la entrega del mismo.

Para comenzar, al menos uno de los integrantes del grupo deberá tener un usuario de GitHub, e ingresar con este usuario a la URL de invitación que será enviada por la lista de correo de la materia. Al entrar, se le pedirá que seleccione su nombre de la lista de alumnos inscriptos en la materia, y que escriba el nombre del grupo. Luego de hacerlo, recibirá acceso a un repositorio privado en GitHub con una copia del código base del trabajo práctico.

Los demás integrantes del grupo pueden ingresar al mismo enlace, y seleccionar el grupo que ya fue creado por el primer integrante para obtener acceso al repositorio. Les recomendamos que aprovechen este repositorio para realizar el trabajo de forma colaborativa, aunque esto no es obligatorio.

El repositorio está configurado para que los casos de test se ejecuten de manera automática cada vez que realicen un *push* a la rama `main`, lo que les permitirá tener un *feedback* inmediato sobre su solución.

Somos conscientes de que para poder utilizar esta herramienta es necesario un conocimiento básico de Git [5], que no es parte del contenido de ninguna materia obligatoria de la carrera. Aún así, consideramos que es una herramienta muy útil para el desarrollo de software, y que vale la pena que la conozcan. En caso de que necesiten ayuda con ella, no duden en consultarlo con la cátedra.

Referencias

[1] Python Software Foundation. *Documentación de Python 3.9*. https://docs.python.org/3.9/.

[2] John E. Hopcroft, Rajeev Motwani y Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 2.ª ed. Addison-Wesley, 2001.

[3] Holger Krekel et al. *Documentación de Pytest*. https://docs.pytest.org/en/.

[4] Python Software Foundation. *Virtual environments*. https://docs.python.org/3.11/library/venv.html.

[5] Git Community. *Documentación de Git*. https://git-scm.com/doc.