Teoría de Lenguajes

Segundo cuatrimestre de 2023

Trabajo práctico: *tlengrep*

Segunda parte

Fecha de entrega: jueves 30 de noviembre de 2023 a las 23:59

Versión del enunciado: 1 (6 de noviembre de 2023)

Para la segunda parte de este trabajo práctico, deberán extender lo entregado en la primera parte para incluir la funcionalidad de parsing de expresiones regulares.

1. Descripción del problema

Se desea modificar el programa construido durante la primera parte del trabajo práctico, de forma que pueda ser invocado sin la opción `-m`. Es decir, al ejecutar

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

el programa deberá parsear la cadena de texto `<expresión regular>`, y traducirla a una instancia de la clase `RegEx`, para luego buscar todas sus apariciones en el archivo `<archivo de texto>` (o en la entrada estándar, si no se provee un valor para dicho argumento) aprovechando la funcionalidad implementada en la primera entrega.

1.1. Sintaxis para expresiones regulares

Hasta ahora, las expresiones regulares con las que venimos trabajando en la materia son objetos abstractos. Para poder representarlas en forma textual, es necesario definir una sintaxis.

Existen distintas variantes sintácticas para expresiones regulares, con sutiles diferencias. Nosotros usaremos una adaptación de la sintaxis para expresiones regulares del lenguaje de programación Perl[3], que es la implementada por la mayoría de los lenguajes de programación modernos.

Las expresiones regulares permitidas por nuestra sintaxis no serán exactamente las mismas que soporta el motor que implementamos previamente: además de los operadores binarios (E_1E_2 y $E_1|E_2$) y unarios (E^*, E^+) definidos en la primera entrega, agregaremos algunos adicionales. Necesitarán pensar cómo expresar estos operadores nuevos en función de los que ya tienen implementados.

1.1.1. Cadena vacía

La expresión regular λ será representada por la cadena vacía (`""`). Para poder usarla como operando de algún operador, será necesario encerrarla entre paréntesis (ver sección 1.1.7).

1.1.2. Caracteres

Los siguientes caracteres estarán reservados para representar operadores: `|`, `*`, `+`, `?`, `\`, `(`, `)`, `[`, `]`. Todos los caracteres no reservados deben interpretarse como la expresión regular correspondiente a dicho carácter; por ejemplo, la cadena `"a"` representa el lenguaje $\{a\}$. Esto incluye a los espacios en blanco (`" "`). Si se quiere usar un carácter reservado como un carácter normal, se debe anteponer `\` como carácter de escape. Por ejemplo, `"\|"` y `"\\"` se interpretan como los caracteres `|` y `\`, respectivamente.

1.1.3. Operadores binarios

Sean `R` y `S` dos cadenas que representan expresiones regulares. Podremos combinar `R` y `S` mediante los siguientes operadores binarios:

Sintaxis	Nombre	Coincide con	Ejemplo
`R|S`	Unión	O bien `R`, o bien `S`	`"a|b"` representa el lenguaje $\{a, b\}$
`RS`	Concatenación	`R` seguida de `S`	`"ab"` representa el lenguaje $\{ab\}$

1.1.4. Operadores unarios

Además, contaremos con los siguientes operadores unarios:

Sintaxis	Nombre	Coincide con	Ejemplo
`R*`	Clausura de Kleene	Cero o más apariciones de `R`	`"a*"` representa el lenguaje $\{\lambda, a, aa, aaa, \ldots\}$
`R+`	Clausura positiva	Una o más apariciones de `R`	`"a+"` representa el lenguaje $\{a, aa, aaa, \ldots\}$
`R?`	Opcional	Cero o una aparición de `R`	`"a?"` representa el lenguaje $\{\lambda, a\}$
`R{n}`	Cuantificador simple	Exactamente n apariciones de `R`	`"a{3}"` representa el lenguaje $\{aaa\}$
`R{n,m}`	Cuantificador doble	Entre n y m apariciones de `R`	`"a{3,5}"` representa el lenguaje $\{aaa, aaaa, aaaaa\}$

Notar que las llaves (`{` y `}`) no son caracteres reservados. Se debe procurar interpretarlas como caracteres normales siempre que *no* están siendo usadas como parte de un cuantificador. Por ejemplo, las cadenas `"{"`, `"{a"` y `"}}"` son expresiones regulares válidas donde las llaves juegan el rol de un carácter normal. Sí está permitido, en cambio, interpretar la cadena `"{3}"` como un error de sintaxis, ya que se trata de un cuantificador mal aplicado.

1.1.5. Clases de caracteres

También contaremos con la posibilidad de definir clases de caracteres. Una clase de caracteres es una expresión de la forma `[`$x_1x_2\ldots x_n$`]`, donde cada x_i puede ser:

- Un carácter cualquiera (no reservado).
- Un rango de caracteres, de la forma `x-y`, donde `x` e `y` son caracteres cualesquiera (no reservados) con `x` anterior a `y` en orden lexicográfico, representando todos los caracteres entre `x` e `y` en dicho orden. Por ejemplo, `"j-n"` representa los caracteres j, k, l, m y n.

Una expresión de este tipo coincide con exactamente *una* aparición de alguno de los caracteres que figuran en su interior, o que están comprendidos en algún rango que figura en su interior. Por ejemplo, la expresión `"[af-i2-5x]"` representa el lenguaje $\{a, f, g, h, i, 2, 3, 4, 5, x\}$, mientras que la expresión `"[a-zA-Z0-9]"` coincide con cualquier carácter alfanumérico. La expresión `"[]"`, por su parte, nos permitirá representar el lenguaje vacío ($\varnothing$).

Por supuesto, las clases de caracteres pueden combinarse con los demás operadores. Por ejemplo, la expresión `"a[dg]{2}"` representa el lenguaje $\{add, adg, agd, agg\}$.

Dado que el guión (`-`) no es un carácter reservado, debe ser interpretado como un carácter normal, salvo que esté siendo usado para definir un rango dentro de una clase de caracteres.

1.1.6. Clases de caracteres especiales

Tendremos dos clases de caracteres especiales, con sintaxis abreviada:

- `\d`: Cualquier dígito decimal. Equivale a `"[0-9]"`.
- `\w`: Cualquier carácter alfanumérico, más el guión bajo (`_`). Equivale a `"[a-zA-Z0-9_]"`.

1.1.7. Precedencia y asociatividad

La siguiente tabla muestra la precedencia y asociatividad de los operadores (ordenados de mayor a menor precedencia):

Operador	Sintaxis	Precedencia	Asociatividad
Cuantificadores	`R*`, `R+`, `R?`, `R{n}`, `R{n,m}`	3	-
Concatenación	`RS`	2	a izquierda
Unión	`R|S`	1	a izquierda

Esto quiere decir que, por ejemplo:

- `"ab|cd"` es la unión de `"ab"` y `"cd"`, y no la concatenación de `"a"`, `"b|c"` y `"d"`.
- En `"ab*"`, el operador `*` se aplica a `"b"`, y no a `"ab"`.

Pueden usarse paréntesis (`(` y `)`) para alterar la precedencia de los operadores. Así, en los ejemplos anteriores, podría haberse escrito `"a(b|c)d"` y `"(ab)*"`, respectivamente.

1.2. Características de la solución

Se espera que implementen un parser (analizador sintáctico) de alguno de los tipos vistos en la materia que sea capaz de reconocer cadenas según la sintaxis recién descripta (e informar de errores de sintaxis), y una traducción guiada por la sintaxis que las convierta a la representación interna correspondiente, basada en subclases de la clase abstracta `RegExp`, para poder ser pasadas al motor de expresiones regulares implementado en la entrega anterior.

En caso de utilizar una gramática ambigua, o que presente algún conflicto para el tipo de parser a utilizar, deberán resolver estos conflictos de forma *explícita*, teniendo cuidado de respetar la precedencia y asociatividad de los operadores.

Recomendamos utilizar la biblioteca PLY (Python Lex-Yacc)[2] para generar el combo de analizador léxico y sintáctico. Aconsejamos también ejecutar la función `yacc`, que genera el parser, con el parámetro `debug=True`. De esta manera, PLY generará un archivo `parser.out`

que contiene información útil para depurar el parser; prestar especial atención a las posibles advertencias sobre conflictos que aparecen al final del archivo.

2. Condiciones de entrega

- El trabajo práctico es grupal y debe realizarse en los **mismos grupos** de tres integrantes formados para la entrega anterior.
- El trabajo debe **complementar** la entrega previa, incorporando tanto las correcciones realizadas a la misma como las nuevas funcionalidades solicitadas en este enunciado.
- La fecha límite de entrega es el jueves 30 de noviembre de 2023 a las 23:59, **sin excepción**.
- La entrega se realizará a través de **GitHub Classroom**, reutilizando los mismos repositorios de la entrega anterior. 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.
- Insistimos con la recomendación enfática de evitar usar herramientas de inteligencia artificial (como GitHub Copilot, ChatGPT, etc.) para asistir en la realización de la solución.

2.1. Código fuente

La entrega deberá incluir el **código fuente** de su solución, que debe extender lo realizado en la entrega anterior y seguir las mismas pautas planteadas en el enunciado de la misma.

El código base provisto por la cátedra incluye un conjunto de casos de prueba para verificar el funcionamiento de las nuevas funcionalidades. Pueden ejecutarlos de la siguiente manera:

```
$ pytest -k test_parser.py
```

Que su solución pase estos casos de prueba es condición necesaria, pero no suficiente, para su aprobación. Recomendamos agregar todos los casos de test adicionales que consideren necesarios.

2.2. Informe

La entrega deberá incluir una segunda parte del **informe** que complemente la anterior, explicando en forma breve la solución adoptada para la entrega, y detallando:

- La gramática elegida para el parser, indicando el tipo de la misma y, de presentar conflictos, cuáles son y cómo fueron resueltos.
- El tipo de parser elegido y, si corresponde, la justificación de dicha elección.
- Cualquier otra decisión de diseño que hayan tomado y consideren relevante documentar.

Referencias

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2.ª ed. Addison-Wesley, 2007.

[2] David M. Beazley. *PLY (Python Lex-Yacc)*. https://www.dabeaz.com/ply/.

[3] Perl Foundation. *Perl regular expressions*. https://perldoc.perl.org/perlre#Regular-Expressions.