Trabajo Práctico - Teoría de Lenguajes

Integrantes

Nombre	Mail	LU
Diego Senarruzza	diegosenarruzza@gmail.com	449/17
Julian Zylber	jzylber@dc.uba.ar	21/18
Manuel Panichelli	panicmanu@gmail.com	72/18

Implementación

La gramática original resulta ser ambigua, frente a esto teniamos dos opciones. O bien definiamos la precedencia de los operadores y su asociatividad (aprovechando las herramientas de precedencia provistas por *ply*), o bien realizabamos una reescritura de la gramática.
Lo elegido fue lo segundo, la gramática modificada resultante es SLR (verificado con grammophone) y expresa en sus producciones la precedencia y asociatividad de los operandos, como se muestra a continuación:

Original

```
<{E}, {|, *, +, ?, ., char, (, )}, P, E>

P:  E → E E
      | E | E
      | E *
      | E +
      | E ?
      | ( E )
      | caracter
      | .
```

Cambiada

```
<{E, C, U, A}, {|, *, +, ?, ., char, (, )}, P', E>

P':    E → E|C | C
       C → CU | U
       U → A* | A+ | A? | A
       A → . | char | (E)
```

Lexer y Parser

El trabajo fue implementado enteramente en **python**, en conjunto con la ayuda de la biblioteca `ply` para el desarrollo del *lexer* y el *parser*. Durante la ejecución del parser, las producciones se encargan de generar y guardar en su nodo padre una objeto que haga de representación del operando o terminal (según corresponda) de la regex que se está parseando. Esto puede entenderse mejor desde el lado de una gramática de atributos en la cual *sintetizamos* este objeto, solo que en lugar de usar un atributo reemplazamos el valor del padre directamente. El resultado final es el de un objeto que sintetiza la expresión regular ingresada.
En caso de haber errores en el lexing o parsing se imprime un error y se detiene la ejecución.

AFD

Utilizando el método de las derivadas construimos el AFD correspondiente. Por simplicidad definiremos el alfabeto como aquellos caracteres que aparezcan en la expresión regular dada por el usuario (en el caso de que aparezca un punto, se agregan todos los caracteres validos al diccionario). Definimos el estado inicial como el objeto generado por la gramática y, a partir de este:

- Se recorre el alfabeto y los estados actuales, realizando una derivacion por símbolo.
- Se realiza una simplificación del estado obtenido por la derivación, siendo esta de:
  - Lambdas concatenados.
  - Vacios concatenados/Vacios en una misma operación Or.
  - Or's redundantes.
- Se guarda el estado en caso de no ser un duplicado de los que ya se tiene (es decir si la expresión regular que representa es distinta a las ya g uardadas).
- Se guarda la transiciones por símbolo en un diccionario.

Notemos que no contamos con un estado trampa, sino que este se deja implicito en el caso de que alguna transición por algún símbolo del alfabeto no exista.

Para terminar, se detectan los estados finales como aquellos que contengan lambda en su lenguaje.

Nota acerca de la construcción del AFD

En consultas con el corrector nos dimos cuenta de que no hay manera de ver si dos expresiones regulares son iguales, sin pasar estas a un autómata. Y por lo tanto, que la simplificación realizada es insuficiente paras solucionar los casos de particulares (siendo el mejor método el de Thompson, pasando el automata resultante a un AFD). Como ya lo estaba hecho, nos dijo que lo dejemos así y que se tenía en cuenta para la corrección.

Busqueda de subcadenas

Como queremos rezliar una busqueda de una subcadena que pertenezca al lenguaje de la expresión regular `R` ingresada por el usuario (en lugar de matchear con una linea entera), generamos un AFD para la expresión `.*(R).*`. Se recorre cada linea del archivo pasado por parámetro y se imprimen por *stdout* aquellas para las cuales el AFD matchea.

Ejecutar

Para bajar las dependencias (solo `ply` y `dataclasses`),

```
pip3 install -r requirements.txt
```

Para correrlo,

```
$ python3 main.py files/telefonos.txt "54 9 11((43)?|(..))(43)+"
54 9 1156434343
54 9 1178434343
54 9 117843434343
```

Para ejecutar los tests,

```
$ python3 tests.py
.
----------------------------------------------------------------------
Ran 1 test in 1.107s

OK
```

Casos de prueba

Se generó un archivo de texto con algunos números de telefono tanto correctos como incorrectos. Generamos distintas regex para comprobar el correcto funcionamiento de los automatas generados.

```
# grep/files/telefonos.txt
54 9 1117428196
54 9 1156434343
54 9 1178434343
54 9 117843434343
cosas antes 54 9 1117428196 cosas despues
54 9 1112469424
5996714627827
59 9 6714627827
5685784939375769
```

- La regex `/54 9 11......../` verifica que empieze como un número de CABA, da como resultado:

  ```
  54 9 1117428196
  54 9 1156434343
  54 9 1178434343
  54 9 117843434343
  cosas antes 54 9 1117428196 cosas despues
  54 9 1112469424
  ```

- La regex `/54 9 11((43)?|(..))(43)+/` filtra el caso anterior por aquellos que contengan el "43", da como resultado:

```
54 9 1156434343
54 9 1178434343
54 9 117843434343
```

- La regex `/-54/` tiene un caracter ilegal, por lo tanto devuelve un error del Lexer:

```
Couldn't parse expression: Lexer: Illegal character '-'
```

- La regex `/54))/` tiene paréntesis desbalanceado, por lo tanto devuelve un error del Parser:

```
Couldn't parse expression: Parser: Syntax error at ')'
```

- La regex `/54()/` tiene una expresión vacía, por lo tanto devuelve un error del Parser:

```
Couldn't parse expression: Parser: Syntax error at ')'
```