01 Explaining Regular Expressions (Lab)

Explain in simple words the languages described by following regular expressions. Avoid paraphrasing the regular expression!

1. `0(0|1)*0`

The set of strings over $\{0, 1\}$ starting and ending with 0 (and therefore of length ≤ 2).

2. `([0]1)*[0]`

The set of strings over {0,1} such that any two 0's are separated by at least one 1.

02 Writing Regular Expressions (Lab)

Assume that the vocabulary is `{a, b}`. Give regular expressions for following languages:

1. All sequences that contain `aa` and contain `bb` !

```
((a|b)*aa(a|b)*bb(a|b)*)|((a|b)*bb(a|b)*aa(a|b)*)
```

2. All sequences that do not begin with `aaa`

```
(b|ab|aab)(a|b)*
```

3. All sequences in which `aa` occurs exactly once. Note that in `aaa`, it would occur twice.

```
([a]b)*aa(b[a])*
```

4. All sequences in which every `a` is either immediately preceded or immediately followed by a `b`, for example `baab`, `aba`, `b`.

```
([a]b[a])*
```

03 Three Formalizations (Lab)

Consider the language `L` of possibly empty strings over the symbols `{a, b, c}` where the first `a` precedes the first `b`.

1. Give a regular grammar that generates `L`.

`G = (T, N, S, P)` where `T = {a, b, c}`, `N = {X, Y}`, `S = X`, and the productions `P` are:

```
X → cX |   | aY
Y →   | aY | bY | cY
```

2. Give a regular expression that describes `L`.

`c*[a(a|b|c)*]`

3. Give a finite state automaton that accepts `L`.

`A = (T, Q, R, q , F)` where `T = {a, b, c}`, `Q = {q , q }`, `F = {q , q }`, and the transitions `R` are:

- `q  c → q`
- `q  a → q`
- `q  a → q`
- `q  b → q`
- `q  c → q`

Note how `A` is constructed from `G`: `q , q` corresponds to `X`, `Y`, for every production there is an equivalent transition, the initial state `q` corresponds to the start symbol `X`, the final states corresponds to nonterminals that have a production to .

05 Explaining Regular Expressions

Explain in simple words the languages described by following regular expressions. Avoid paraphrasing the regular expression!

1. `(a*b*)*`

A language made up of strings that consist of any number of repetitions of the substring "a" followed by any amount of "b"s or both.

2. `(a*ba*b)*a*`

A language made up of strings that consist of any amount of "a" followed by a single "b", followed by any amount of "a"s, followed by a single "b" again, and finally ending with zero or multiple "a"s.

3. `(a*[ba*c])*`

A language comprised of a set of strings which consists of `a`s before any sequence of `bac` where there is any amount of `a`s in the string, or an empty set.

06 Writing Regular Expressions

Assume that the vocabulary is `{a, b}`. Give a regular expression that describes all sequences that must contain at least two consecutive occurrences of `a`, like `aa`, `baa`, `aaa`, `abaaaaba`!

`b*aaa*b*a*`

Now give a regular expression of the complement of above language (without using a complement operator), that is those sequences over `{a, b}` that must not contain two or more consecutive occurrences of `a`, but still may have an arbitrary number of occurrences of `a`!

```
(b*ab)*
```

Assume that the vocabulary is `{a, b, c}`. Give a regular expression that describes all sequences in which the number of `a` symbols is divisible by three.

```
(b|c)*|((b|c)*a(b|c)*a(b|c)*a(b|c)*)*
```

Now give a regular expression for sequences with the total number of `b` and `c` symbols being three.

```
a*(b|c)a*(b|c)a*(b|c)a*
```

07 Regular Grammar to Regular Expression

Here is a regular grammar in EBNF for fractional numbers:

```
N → '0' N | ... | '9' N | '.' F
F → '0' D | ... | '9' D
D → '0' D | ... | '9' D | ''
```

Transform the grammar into a regular expression. Use equalities to simplify the regular expressions. Give every step of the transformation in detail and state the rule that you are applying.

`F` equivalent production:

```
F → ('0'|...|'9')D
```

`D` equivalent production:

```
D → ('0'|...|'9')D | ''
```

Using Arden's rule:

```
D → ('0'|...|'9')* ''.
```

`D` elimination by substitution into `F`:

```
F → ('0'|...|'9')('0'|...|'9')* ''
```

Equivalent production for `N`:

```
N → ('0'|...|'9')N |'.' F
```

By Arden's rule, we simplify:

```
N → ('0'|...|'9')* '.' F
```

`F` elimination by substituting into `N`:

```
N → ('0'|...|'9')* '.' ('0'|...|'9')('0'|...|'9')* ''
```

Final equivalent regular expression:

```
('0'|...|'9')* '.' ('0'|...|'9')* ''
```

08 Testing Regular Expression

Using the notation from the course notes, write a regular expression for identifiers: an identifier is a sequence of letters `abcdefghijklmnopqrstuvwxyz` and digits `0123456789` starting with a letter. You may use abbreviations

```
(a|...|z)((a|...|z)*(0|...|9)*)*
[a-z]+ ([a-z]* | [0-9]*)*
```

```
class FiniteStateAutomaton:
    def __init__(self, T, Q, R, q0, F):
        self.T, self.Q, self.R, self.q0, self.F = T, Q, R, q0, F
    def __repr__(self):
        return str(self.q0) + '\n' + ' '.join(self.F) + '\n' + \
               '\n'.join(r[0] + ' ' + r[1] + ' → ' + r[2] for r in self.R)
```

```
class Choice:
    def __init__(self, e1, e2): self.e1, self.e2 = e1, e2
class Conc:
    def __init__(self, e1, e2): self.e1, self.e2 = e1, e2
class Star:
    def __init__(self, e): self.e = e
```

```
def string(s: set) -> str:
    return '{' + ', '.join(e for e in s) + '}'


def deterministicFSA(fsa: FiniteStateAutomaton) -> FiniteStateAutomaton:
    q 0 = frozenset({fsa.q0})
    Q , R , visited = {q 0}, set(), set()
    # print(Q , R , visited)
    while visited != Q :
        q  = (Q  - visited).pop(); visited |= {q }
        for t in fsa.T:
            r  = {r for (q, u, r) in fsa.R if u == t and q in q }
            if r  != set(): Q  |= {frozenset(r )}; R  |= {(q , t, frozenset(r ))}
        # print(Q , R , visited)
    F  = {q  for q  in Q  for f in fsa.F if f in q }
    return FiniteStateAutomaton(fsa.T, {string(q ) for q  in Q },
        {(string(q ), t, string(u )) for (q , t, u ) in R },
        string(q 0), {string(f ) for f  in F })
```

```
def accepts(fsa: FiniteStateAutomaton,  : str) -> bool:
      = {(q, a): r for (q, a, r) in fsa.R}
    q = fsa.q0
    for t in  :
        if (q, t) in  : q =  [q, t]
        else: return False
    return q in fsa.F

def REToFSA(re):
    global QC
    if re == '': q0 = str(QC); QC +=1; return FiniteStateAutomaton(set(), {q0}, set(), q0,
    elif type(re) == str:
        q0 = str(QC); QC +=1; q1 = str(QC); QC += 1
        return FiniteStateAutomaton({re}, {q0, q1}, {(q0, re, q1)}, q0, {q1})
    elif type(re) == Choice:
        A1, A2 = REToFSA(re.e1), REToFSA(re.e2)
        R2 = {(A1.q0 if q == A2.q0 else q, a, r) for (q, a, r) in A2.R} # A2.q0 renamed to
        F2 = {A1.q0 if q == A2.q0 else q for q in A2.F} # A2.q0 renamed to A1.q0 in A2.F
        return FiniteStateAutomaton(A1.T | A2.T, A1.Q | A2.Q, A1.R | R2, A1.q0, A1.F | F2)
    elif type(re) == Conc:
        A1, A2 = REToFSA(re.e1), REToFSA(re.e2)
        R = A1.R | {(f, a, r) for (q, a, r) in A2.R if q == A2.q0 for f in A1.F} | \
            {(q, a, r) for (q, a, r) in A2.R if q != A2.q0}
        F = (A2.F - {A2.q0}) | (A1.F if A2.q0 in A2.F else set())
        return FiniteStateAutomaton(A1.T | A2.T, A1.Q | A2.Q, R, A1.q0, F)
    elif type(re) == Star:
        A = REToFSA(re.e)
        R = A.R | {(f, a, r) for (q, a, r) in A.R if q == A.q0 for f in A.F}
        return FiniteStateAutomaton(A.T, A.Q, R, A.q0, {A.q0} | A.F)
    else: raise Exception('not a regular expression')

def convertRegExToFSA(re):
    global QC; QC = 0
    return REToFSA(re)
```

Test your answer by expressing it with Python constructors `Choice`, `Conc`, `Star` and calling it `I`.

```
def selectString(y):
    result = y[0]
    for x in range(1, len(y)):
        result = Choice(result, y[x])
    return result

Z = selectString("abcdefghijklmnopqrtuvwxyz")
X = selectString("0123456789")
Y = Star(Conc(Star(Z), Star(X)))
I = Conc(Z,Y)

A = deterministicFSA(convertRegExToFSA(I))
assert accepts(A, 'cloud7')
assert accepts(A, 'if')
assert accepts(A, 'b12')
assert not accepts(A, '007')
assert not accepts(A, '15b')
assert not accepts(A, 'B12')
assert not accepts(A, 'e-mail')
```

09 Finding E-mail Addresses

For your graduation party you like to invite all your friends of whom you have either an e-mail address or a telephone number. As you never had time to keep an address book, you like to search for these in all your files using `grep`. In the cells below, write `grep` commands. The `%%bash` cell magic runs the cell in the bash shell; the `%%capture output` cell magic captures the output of the cell in the Python variable `output`.

1. E-mail addresses start with one or more upper case letters `A-Z`, lower case letters `a-z`, and symbols `+-._`, followed by the `@` sign, followed by a domain. The domain is sequence of subdomains separated by `.`, where each subdomain consists of a number upper and lower case letters, digits, and symbol `-`. There have to be at least two subdomains (i.e. one `.`). The last subdomain is called the top-level domain and must consists only of two to six upper or lower case letters. E-mail addresses must start at the beginning of a line or after a separator and must end at the end of a line or a separator. Write a shell command using `grep` that, from the directory in which it is started, recursively visits all subdirectories and prints those lines of files that contain an e-mail address. Use `\b` as the separator.

```
%%capture output
%%bash
# grep -rEo "\b[A-Za-z0-9+._-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,6}\b" \data
grep -rE '\b[A-Za-z+-._]+@[A-Za-z0-9-]+.[A-Za-z]{2,6}\b' \data
```

```
print(output) # for testing
```

The regular expression pattern being searched for is enclosed in single quotes `'...'` and consists of three parts separated by the `@` symbol:

1. `\b[A-Za-z+-._]+` - matches the username of the email address. `\b` specifies a word boundary and `[A-Za-z+-._]+` matches one or more alphabetical letters, plus sign, hyphen, period or underscore characters.

2. `@[A-Za-z0-9-]+.` - matches the at symbol (@) and the domain name of the email address. `[A-Za-z0-9-]+` matches one or more alphabetical letters or digits, and hyphen characters. The dot (.) outside the square brackets matches a literal dot character.

3. `[A-Za-z]{2,6}\b` - matches the top-level domain of the email address, such as com, edu, gov, etc. `[A-Za-z]{2,6}` matches 2 to 6 alphabetical

letters and the `\b` specifies another word boundary.

```
assert str(output) == """data/03/friends.txt:abcd@abc.ca
data/03/friends.txt:abcde@ab-BC.com
data/03/friends.txt:@e-mail@add.ress@
data/02/other-friends.txt:ABCabc+-._@ancbd.ca
data/02/other-friends.txt:ABCabc+-._@mcmaster.io.ca
data/02/other-friends.txt:ABCabc+-._@school.image
data/02/other-friends.txt:ABCabc+-._@school3-computer.image
data/02/other-friends.txt:ABCabc+-._@school3-IT.image.tor.chrome.ca
data/02/other-friends.txt:ABCabc+-._@school3-IT.image.tor.chrome.canadannn
data/02/other-friends.txt:ABC123abc+-._@school3-IT.imageal.tor.chrome.canadannn
data/01/friends.txt:Marion Floyd (905 263-7740 jpflip@yahoo.com
data/01/friends.txt:Cora Larson (905) 255-8305 frederic@yahoo.ca
data/01/friends.txt:Van Craig 905) 608-2616 chunzi@aol.com
data/01/friends.txt:Emilio Morrison (905) 2877753 cantu@sbcglobal.net
data/01/friends.txt:Ismael Hanson (905) 755 9372 satch@hotmail.com
data/01/friends.txt:Wayne Douglas (905)222-3316 tfinniga@verizon.net
data/01/friends.txt:Tomas Carlson (905746-0359 ardagna@me.com
data/01/friends.txt:Laurence Newman 9057803232 jaarnial@icloud.com
data/01/friends.txt:Lori Sherman 905-543-7753 chaki@att.net
data/01/friends.txt:Gladys Brock (539) 728-2363 lukka@icloud.com
"""
```

You are looking for telephone numbers in the `905` area for your party. Valid numbers are of the form `(905) 123 4567`, `(905) 1234567`, `905-123-4567`. However, `9051234567`, as well as `905) 123 4567`, `905-123 4567`, are not. Telephone addresses must start at a the beginning of a line or after a separator and must end at the end of a line or a separator. Write a shell command using grep that, from the directory in which it is started, recursively visits all subdirectories and prints those lines of files that contain a telephone number.

```
%%capture output
%%bash
# grep -rE '^(\(905\) [0-9]{3} [0-9]{4})|(\(905\) [0-9]{7})|(905-[0-9]{3}-[0-9]{4})' \data
grep -Er '(\(905\) [0-9]{3} [0-9]{4})|(\(905\) [0-9]{7})|905-[0-9]{3}-[0-9]{4}' \data
```

```
print(output) # for testing
```

This command searches for specific phone number patterns in files under the directory named "data" using the **`grep`** command with the following options:

- `-E` option enables the extended regular expression syntax which allows us to use the `|` character to specify multiple patterns.
- `-r` option enables recursive search, which means it will search for files in subdirectories as well.

The regular expression pattern being searched for is enclosed in single quotes `'...'` and consists of three parts separated by the `|` character:

1. `(\(905\) [0-9]{3} [0-9]{4})` - matches phone numbers in the format `(905) xxx xxxx`, where `x` can be any digit from 0 to 9. The backslashes are used to escape the parentheses and prevent them from being interpreted as special characters by the shell.
2. `(\(905\) [0-9]{7})` - matches phone numbers in the format `(905) xxxxxxx`, where `x` can be any digit from 0 to 9.
3. `905-[0-9]{3}-[0-9]{4}` - matches phone numbers in the format `905-xxx-xxxx`, where `x` can be any digit from 0 to 9.

```
assert str(output) == """data/03/friends.txt:(905) 123 4567
data/03/friends.txt:(905) 1234567
data/03/friends.txt:905-123-4567
data/02/other-friends.txt:(905) 123 4567
data/02/other-friends.txt:(905) 1234567
data/02/other-friends.txt:905-123-4567
data/01/friends.txt:Emilio Morrison (905) 2877753 cantu@sbcglobal.net
data/01/friends.txt:Ismael Hanson (905) 755 9372 satch@hotmail.com
data/01/friends.txt:Lori Sherman 905-543-7753 chaki@att.net
"""
```

10 Extracting Columns from CSV

Consider a file with columns separated by a single comma. Write an `sed` command to extract and output only the first column. Apply your command to the file `data/q.csv`:

```
%%writefile data/q.csv
a,b,c
d,e,f
gh,i,jkl
m n o,pq r,stuv1
```

The `%%bash` cell magic runs the cell in the bash shell; the `%%capture output` cell magic captures the output of the cell in the Python variable `output`.

```
%%capture output
%%bash
#sed -r 's/TODO/TODO/g' data/q.csv
sed 's/,.*//' data/q.csv
```

```
print(output) # for testing
```

Here is an explanation of the command:

- `s/,.*//`
- `s/` - start of the substitution command
- `,` - matches a comma character

- `.*` - matches any number of characters after the comma
- `//` - replaces the matched text with nothing (i.e., deletes it)

```
assert str(output) == """a
d
gh
m n o
"""
```

Write an `sed` command to extract and output the second column

```
%%capture output
%%bash
sed 's/[^,]*,\([^,]*\),.*/\1/' data/q.csv
print(output) # for testing
```

This command uses the `sed` command to manipulate text in a file named `q.csv` under the `data` directory. The `sed` command is a stream editor that can be used to modify text.

The `s` command in `sed` stands for "substitute". The regular expression pattern inside the first pair of forward slashes `/.../` matches a specific pattern in the text, and the second part after the second forward slash is the replacement text.

Here is an explanation of the command:

- `s/[^,]*,\([^,]*\),.*/\1/`
- `s/` - start of the substitution command
- `[^,]*,` - matches any character that is not a comma (`,`) followed by a comma. The asterisk (`*`) matches zero or more occurrences of the preceding character class.
- `\([^,]*\),` - matches any character that is not a comma (`,`) between two commas and captures it in a group. The parentheses (`\(` and `\)`) define a capture group that can be referred to later with `\1`.
- `.*` - matches any number of characters after the captured group.
- `\1` - replaces the entire matched text with the captured group.

Therefore, the command will search for a pattern in the `q.csv` file where there are three comma-separated fields and replace the entire line with the second field of each line. The command assumes that the second field of each line does not contain any commas.

```
assert str(output) == """b
e
i
pq r
"""
```

Write an `sed` command to extract and output the third column

```
%%capture output
%%bash
sed 's/^\([^,]*\),\([^,]*\),\([^,]*\).*/\3/' data/q.csv

# Explanation:

# ^ matches the beginning of a line.
# [^,]* matches zero or more characters that are not a comma.
# \(...\) creates a capturing group.
# , matches a single comma.
# .* matches any number of characters until the end of the line.
# \3 outputs the contents of the third capturing group.
```

```
print(output) # for testing
```

Here is an explanation of the command:

- `s/^\([^,]*\),\([^,]*\),\([^,]*\).*/\3/`
- `s/` - start of the substitution command
- `^` - matches the beginning of the line
- `\([^,]*\)` - matches any character that is not a comma (`,`) and captures it in a group. The parentheses (`\(` and `\)`) define a capture group that can be referred to later with `\1`, `\2`, or `\3`, depending on which group is being referred to.
- `,` - matches a comma character
- `\([^,]*\)` - matches any character that is not a comma (`,`) and captures it in a group.
- `,` - matches another comma character
- `\([^,]*\)` - matches any character that is not a comma (`,`) and captures it in a group.
- `.*` - matches any number of characters after the third captured group.
- `\3` - replaces the entire matched text with the third captured group.

Therefore, the command will search for a pattern in the `q.csv` file where there are three comma-separated fields at the beginning of each line and replace the entire line with the third field of each line. The command assumes that each line in the `q.csv` file has exactly three comma-separated fields at the beginning of the line.

```
assert str(output) == """c
f
jkl
stuv1
"""
```

11 Lowercasing HTML tags (sed)

Consider `IMG` tags in HTML files, as in:

```
%%writefile data/q.html
<img src="PiCtuRe.PnG "/>
<img src="PiCtuRe.PnG"></img>
<img src="PiCtuRe.PnG">alt</img>
<img src="PiCtuRe.PnG"> alt   </img>
<img src ="PiCtuRe.PnG" />
<img src = "PiCtuRe.PnG"/>
<img onclick="alert('Clicked!')" src = "PiCtuRe.PnG"/>
```

Write an `sed` command to lowercase `SRC` values. For example, `<img src="PiCtuRe.PnG">` should be replace by `<img src="picture.png">`. Note that there can be arbitrary space around src and = and before >.

```
%%capture output
%%bash
#sed -r 's/TODO/TODO/g' data/q.html
# sed 's/\(<img[^>]*src *= *"\)[^"]*\("[^>]*>\)/\1\L&\E\2/' data/q.html
sed 's/\(<img[^>]*src *= *\)"\([^"]*\)"/\1"\L\2"/Ig' data/q.html
print(output) # for testing
```

Here is an explanation of the command:

- `s/` - start of the substitution command
- `\( ... \)` - defines a capture group, which can be referred to later with `\1`
- `<img[^>]*src *= *"` - matches an `img` tag with an `src` attribute, and captures everything up to the opening double quote of the `src` attribute in the first capture group. The `[^>]*` matches any characters except the `>` character, which ends the opening `img` tag. The `src *= *"` matches the `src` attribute name and any number of spaces before the opening double quote of the attribute value.
- `"\([^"]*\)"` - matches the value of the `src` attribute inside double quotes, and captures it in the second capture group.
- `/` - separates the search pattern from the replacement pattern
- `\1` - replaces the entire matched text with the first capture group, which is everything before the opening double quote of the `src` attribute.
- `"` - inserts a double quote character in the replacement pattern
- `\L\2` - converts the second capture group (the value of the `src` attribute) to lowercase using the `\L` command, and inserts it in the replacement pattern.
- `"` - inserts another double quote character in the replacement pattern
- `/` - end of the substitution command

Therefore, the command will search for all `img` tags with an `src` attribute in the `q.html` file, and replace the value of the `src` attribute with a lowercase version of itself. The opening double quote of the `src` attribute value is preserved, and the closing double quote of the `src` attribute value is replaced with a new closing double quote. Any spaces around the equals sign between the `src` attribute name and value are also preserved.

```
assert str(output) == """<img src="picture.png "/>
<img src="picture.png"></img>
<img src="picture.png">alt</img>
<img src="picture.png"> alt   </img>
<img src ="picture.png" />
<img src = "picture.png"/>
<img onclick="alert('Clicked!')" src = "picture.png"/>
"""
```