CISS 445: Programming Languages
Assignment 7

Objectives:

- Design DFA and NFA
- Write regular expressions using Perl-style regular expression syntax

Mathematically given a set of symbols S, a regular expression is a string made up of characters from S together with U (union), ((left parenthesis),) (right parenthesis), * (Kleene star), ε (empty string), Ø (empty set).

Although these characters are enough to mathematically construct regular expressions (and hence any regular language), programmers have found this regular expression syntax cumbersome. Different programming languages and different third party software (classes, modules, packages, etc) have included additional symbols to help in writing regular expressions more compactly. One such syntax is the Perl regular expression syntax. The objective of this assignment is to learn to write regular expression using Perl's regular expression syntax. Note that this is not the only regular expression syntax. In your professional life (in academia or industry) you will come across different regular expression engines.

The following includes a quick introduction to Python. You will need to read the documents listed below for more information. Note that I am using Python 2.7.

Create a directory a07 and in a07 create a07/a07q01, etc. Put your files for a07q01 in a07/a07q01, etc. Tar and gzip your a07 and email it to yliow.submit@gmail.com with subject line “ciss445 a07”. You must email using your college email account.

How to specify DFA and NFA in the questions

Suppose for a question, you have designed an NFA, say you call it N. The states are A, B, C, D, E. Suppose your start state is A and C,D are accept states. Suppose further that the alphabet of your N is {0, 1}. In your N, you have a transition from A to B labeled 0, another transition from A to A labeled 1, and a transition from B to C labeled ε. Then your NFA (say you call it N) is described in main.py as follows:

```
# Name: John Doe
# File: main.py

N = NFA(alphabet=["0", "1"],
        states=["A", "B", "C", "D", "E"],
        start="A",
        accepts=["C", "D"],
        transitions=[("A", "0", "B"),
                     ("A", "1", "B"),
                     ("B", "" , "C")])
```

Note in particular you should use `""` as ε.

Suppose for another question, you have designed a DFA, say you call it M. For instance if your DFA has only two states “q0” and “q1” where “q0” is the start state and “q1” is the accept state, the alphabet is “0” and “1”. As for transitions, “q0” transitions to “q0” on “0”, “q0” transitions to “q1” on “1”, “q1” transitions to “q0” on “0”, and “q1” transitions to “q1” on “1”, then your DFA is described follows in main.py:

```
# Name: John Doe
# File: main.py

M = DFA(alphabet=["0", "1"],
        states=["q0", "q1"],
        start="q0",
        accepts=["q1"],
        transitions=[("q0", "0", "q0"),
                     ("q0", "1", "q1"),
                     ("q1", "0", "q0"),
                     ("q1", "1", "q1")])
```

WARNING: This is a DFA. Therefore out of every state, there are exactly 2 transitions, one labeled 0 and one labeled 1.

Q1. [Design NFA]

Design an NFA N that accept strings made up of symbols 0 and 1 where each string contains an even number of 0s or odd number of 1s.

Complete the following skeleton:

```
# Name: John Doe
# File: main.py

N = NFA(alphabet=[],
        states=[],
        start=None,
        accepts=[],
        transistions=[])
```

Q2. [Design DFA]

Construct a DFA M such that M accepts the same language as the NFA N in Q4. The answer must be entered into string M below using the same format as described in Q4.

Complete the following skeleton:

```
# Name: John Doe
# File: main.py

M = DFA(alphabet=[],
        states=[],
        start=None,
        accepts=[],
        transitions=[])
```

Programming tool

To test your regular expressions, we will use the Python programming language. You should use one of my fedora virtual machines.

```
$ python
Python 2.7.17 (default, Oct 21 2019, 17:19:01)
[GCC 8.3.1 20190223 (Red Hat 8.3.1-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can also download other Python implementation from Python's web site: http://www.python.org. If you're using the Windows version, you can get to the same Python shell when you select the menu item “Python (command line)” from your Start menu under Python.

If you have a Linux box, Python is probably already installed. You can also run the python interpreter by typing python at the shell prompt.

For spring 2020, make sure you have Python 2.7. Do NOT use a 3.x version. You can find all the releases for Python at http://www.python.org. Choose the right one. For classes after fall 2020, make sure you use Python 3.

After running the Python interpreter, you can issue Python statements at the Python prompt in the Python shell:

```
>>> 1 + 1
2
>>>
```

Note that the text in bold is entered by the user. Note that Python responded by printing the resulting value. You can also issue a print command if you like but the two have the same effect:

```
>>> print 1 + 1
2
>>>
```

You can write write your Python statements in a file, say test.py:

```
print 1 + 1
```

After saving it, you can run it at your shell prompt by typing:

```
python test.py
```

Your shell will run the program with python and respond with:

```
2
```

For more information the Python programming language refer to the tutorials and guide at http://www.python.org. Here's one: http://docs.python.org/tutorial/index.html.

Python Strings

Let's talk about Python strings. Python string have the “usual” behavior. For instance you can concatenate strings. Run this:

```
s = "abc"
t = "def"
u = s + t
print u
```

Of course you can cut a piece of the string. Run this:

```
s = "abcdef"
t = s[2]
print t
```

What do you get? Python does not have the concept of characters in the sense of C/C++: a character in Python in actually just a string of length 1. Therefore the variable t above is a string.

Somewhat more surprising is this (if you have not seen this before):

```
s = "abcdef"
t = s[1:3]
print t
```

Instead of 1:3 try different values such as 0:2, 1:4, etc.

Python strings are immutable. In other words you cannot change the contents of the string. For instance if you want to change the first character of a string to x, then you basically have to build a new string like this. Try this:

```
s = "abcdef"
t = "x" + s[1:]
print t
```

You can of course compare strings. Try this:

```
s = "abcdef"
t = "abcdef"
u = "abcdez"
print s == t
print s == u
```

And of course we must have the string length function:

```
s = ""
t = "a"
u = "abcdef"
print len(s), len(t), len(u)
```

For more on Python strings, refer to the tutorial stated above.

Python Functions

Writing a Python function is easy. Here's an example. Run it:

```
def f():
    return 42

print f()
```

And here's another. Run it:

```
def g(x):
    y = "you gave me " + x
    return y

print g("hello world")
```

And another. Run it:

```
def h(x):
    if x < 0:
        a = 0
        b = 1
        c = 2
    elif x == 0:
        a = 1
        b = 2
        c = 3
    else:
        a = 2
        b = 3
        c = 4
    return a + b + c

print h(-1)
print h(0)
print h(2)
```

Note that Python determines blocks by whitespaces. The above also show you how to write branching statements. As for the for-loop, try this:

```
for a in [1,2,3,4,5]:
    print a

for a in range(1, 10):
    print a

for a in range(1, 10, 2):
    print a
```

And for the while-loop try this:

```
s = raw_input('gimme something ... ')
while s != '':
    print s
    s = raw_input('gimme something ... ')
```

Python Regular Expressions

For more information, you will need to study http://www.amk.ca/python/howto/regex/.

Python comes with a regular expression module. With the library you can build a regular expression object. With this object you can carry out “matching” and “searching”: with matching you will get a match object and with searching you will get a search object. We'll talk about matching first.

Examples. The following example shows you how to use the regular expression module in Python. Run this:

```
>>> import re
>>> p = re.compile("abc")
>>> p.match("abc")
<_sre.SRE_Match object at 0x7ff2a090>
>>> p.match("xyz")
>>>
```

In the above example p is a regular expression object corresponding to the regular expression “abc”.

Now we test the regular expression against some strings.

`p.match("abc")` returns an object `<_sre.SRE_Match object at 0x7ff2a090>` telling you that the string `"abc"` matches the regular expression object `"abc"`.

On the other hand when you test the string `"xyz"` against your regular expression, you get nothing. This tells you that `"xyz"` does not match the regular expression of `p`.

However there is a difference between Python's regular expression match functionality and the mathematical regular expression matching that you should be aware of. The regular expression object's `match()` method actually matches leftmost ***substrings*** in the test string. For instance try this (continuing the above example):

```
>>> p.match("abcxyz")
<_sre.SRE_Match object at 0x7ff2a090>
```

As you can see, `"abcxyz"` matches `"abc"` in the sense that `"abc"` appears as a leftmost substring in the test string `"abcxyz"`.

So what if do not want to match leftmost substrings – you want exact match? You can force matching of end-of-string with \Z, the end-of-string syntax. In other words you use the regular expression “`"abc\Z". C`ontinuing the above example, let's build another regular expression object:

```
>>> p1 = re.compile("abc\Z")
>>> p1.match("abc")
<_sre.SRE_Match object at 0x7ff354f0>
>>> p1.match("abcxyz")
```

Note that now `"abcxyz"` does ***not*** match the regular expression `"abc\Z"`. That's because `x` does not match `\Z` since `\Z` is the end-of-string marker.

(Note that you only need to do “import re” once in your Python shell. You don't have to do it again in this example. The only time you need to do that is when you start the Python shell.)

Now let's look closely at a match object. This again:

```
>>> p = re.compile("abc")
>>> match = p.match("abcdef")
```

This gives you a match object, match. You can also view the attributes (including the methods) of the match object:

```
>>> print dir(match)
['__copy__', '__deepcopy__', 'end', 'expand', 'group', 'groupdict',
'groups', 'span', 'start']
```

You print the string that was matched:

```
>>> print match.group()
abc
```

You can print the starting index and ending index of the match within the string “abcdef”:

```
>>> print match.start()
0
>>> print match.end()
3
```

Refer to Python documentation for more information.

Now for searching. While matching attempts to find a pattern starting at the beginning of a string, search tries to match any substring. Try this

```
>>> p2 = re.compile("abc")
>>> search = p2.search("12 34abc56789")
>>> print search
<_sre.SRE_Match object at 0x00A90838>
```

This tells you that the regular expression `"abc"` is found in the string `"12 34abc56789"`, not necessarily at the beginning. We can find what is the substring within `"12 34abc56789"` that matches our regular expression `"abc"` and we can also find the starting index and ending index of this substring within `"12 34abc56789"`:

```
>>> print search.group()
abc
>>> print search.start()
5
>>> print search.end()
8
```

As you can see the `"abc"` is found within `"12 34abc56789"` at starting index position 5 and ending index position 8.

Perl Regular Expression Syntax

There are special characters to denote special meanings. For instance you will see that `[` is used to mean something. But what if you what to match the character `[`? You use the escape character, i.e. you use `\[`.

The only time when you do not need to use the escape character \ is in [...]. The [...] notation is similar to the set notation. You can also specify a range of values in [...]. Here are some examples:

[abcd] matches 'a' or 'b' or 'c' or 'd'.
[a-d] matches 'a' to 'd' (the order is taken from the ASCII table).
[0-9a-z] matches '0' to '9' or 'a'-'z'

[^abc] matches everything except a,b,c

All these special characters used in building a regular expression in Python actually comes from another language: Perl. (There are many other regular expression syntax, but Perl is probably the most famous.)

Therefore the regular expression “[a-d][0-9]\Z” should match strings such as b0, d9:

```
>>> p2 = re.compile("[a-d][0-9]\Z")
>>> p1.match("ad")
>>> p1.match("a0")
```

The regular expression “[a-d][^a-z]” should match “a0” but not “ab”:

```
>>> p3 = re.compile("[a-d][^a-z]")
>>> p3.match("a0")
<_sre.SRE_Match object at 0x7ff354f0>
>>> p3.match("ab")
```

Getting the hang of it yet?

a* matches any number of a (including none)
a+ matches any positive number of a. This is the same as aa*
a{4} same as aaaa
a{2,4} aa or aaa or aaaa
a{2,} aa or aaa or aaaa or aaaaa or aaaaaa or ....
a? same as empty string or a, i.e. optional a. This is the same as a{0, 1}

Try these:

```
>>> p4 = re.compile("[01]{1,2}\Z")
>>> p4.match("0")
<_sre.SRE_Match object at 0x7ff35598>
>>> p4.match("1")
<_sre.SRE_Match object at 0x7ff354f0>
>>> p4.match("00")
<_sre.SRE_Match object at 0x7ff35598>
>>> p4.match("01")
<_sre.SRE_Match object at 0x7ff354f0>
>>> p4.match("10")
<_sre.SRE_Match object at 0x7ff35598>
>>> p4.match("11")
<_sre.SRE_Match object at 0x7ff354f0>
>>> p4.match("000")
>>> p4.match("101")
>>>
```

And this:

```
>>> p5 = re.compile("[01]{2,}")
>>> p5.match("0")
>>> p5.match("00")
<_sre.SRE_Match object at 0x7ff35598>
>>> p5.match("000")
<_sre.SRE_Match object at 0x7ff354f0>
>>> p5.match("0000")
<_sre.SRE_Match object at 0x7ff35598>
>>>
```

You should try a few search examples, printing the group(), start(), and end().

This is not too surprising by now:
a|b   match a or b

```
>>> p6 = re.compile("(ab)|(de)\Z")
>>> p6.match("ab")
<_sre.SRE_Match object at 0x7ff30578>
>>> p6.match("de")
<_sre.SRE_Match object at 0x7ff30260>
```

Note that matches are “greedy” for instance the string “aaaaa” will not match the regular expression “a{3,5}aa\Z” since “aaaaa” will match “a{3,5}” leaving nothing for “aa\Z”.

FAQ

Q: “What about epsilon?”
A: Use an empty substring. Study the following example:

```
[student@localhost ~]$ python
Python 2.7.13 (default, Dec  1 2017, 09:21:53)
[GCC 6.4.1 20170727 (Red Hat 6.4.1-1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import re
>>> p = re.compile("abc(1|2|)def")
>>> p.search("a")
>>> print p.search("a")
None
>>> print p.search("ab")
None
>>> print p.search("abc")
None
>>> print p.search("abc1")
None
>>> print p.search("abc1d")
None
>>> print p.search("abc1de")
None
>>> print p.search("abc1def")
<_sre.SRE_Match object at 0x7f774515c738>
>>> print p.search("abc2def")
<_sre.SRE_Match object at 0x7f774515c738>
>>> print p.search("abcdef")
<_sre.SRE_Match object at 0x7f774515c738>
>>>
```

Q3. [Regex using PERL Syntax]

Write down the mathematical description of a regular expression for strings representing polynomials with integer coefficients. The following are examples of such strings:

- 0
- -2
- 2
- 1+x
- 1 + x
- x + 1
- x+1
- -1+x^3
- 1-x^3
- 1+x+x+x
- 1+x^23+x^3   - 42 x^100
- 0x^100 + -1x
- x^0

Integer coefficients and powers must be integers. An integer is either 0 or a non-zero digit followed by any number of digits. The following are strings which should not be accepted:

- x^
- ^3
- 2^2
- x^x

Here's a skeleton (skeleton code is to help you – it is not meant to be correct or error-free):

```
# Name: John Doe
# File: main.py

import re

p = re.compile("x*\Z") // replace string with correct regex
s = raw_input()
print p.match(s)
```

(The regular expression is for any number of x. Therefore it's not correct although it does match "x" correctly.)

Q4. [Regex using PERL Syntax]

Write a regular expression for strings of 0's and 1's where all maximal substrings of 0's have even length. For instance

- 0000
- 1001100111
- 0011

are accepted by the regular expression. But the following are not:

- 000
- 1001100011
- 100110010

Skeleton code:

```
# Name: John Doe
# File: main.py

import re

p = re.compile("x*\Z") // replace string with correct regex
s = raw_input()
print p.match(s)
```

Q5. [Extracting data from XML using regex]

An XML string is a string in a special format. Here's an XML fragment:

```
<firstname>John</firstname>
```

Here's another:

```
<lastname>Doe</lastname>
```

Here’s another

```
<person>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
</person>
```

or maybe

```
<person id=1231235>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
</person>
```

XML data provides “meaning” and “structure” to otherwise flat data without meaning.

For this question, you will need to visit https://openweathermap.org/. Go ahead and create an account at that website. (You can also go to wikipedia and read up on openweathermap.) After confirming the account, login and click on API (near the top) and then subscribe to the weather data. You should get an email from the website that looks like this:

> Dear Customer!
>
> Thank you for subscribing to Free OpenWeatherMap!
>
> API key:
> - Your API key is **014f77b0g63693crffcbhab9dfvb7903**
> - Within the next couple of hours, it will be activated and ready to use
> - You can later create more API keys on your account page
> - Please, always use your API key in each API call
>
> [etc.]

You will need the above API key.

Wait for a couple of minutes. Make sure you connected to the internet. Then run the following python program:

```
city_id = 4575352
api_key = "014f77b0g63693crffcbhab9dfvb7903"
url = "http://api.openweathermap.org/data/2.5/weather?id=%s&mode=xml&units=imperial&APPID=%s"
url = url % (city_id, api_key)
xml = urllib2.urlopen(url).read()
print xml
```

You will see this:

```
<?xml version="1.0" encoding="UTF-8"?>
<current><city id="4575352" name="Columbia"><coord lon="-81.03"
lat="34"></coord><country>US</country><timezone>-18000</timezone><sun rise="2020-03-
07T11:44:40" set="2020-03-07T23:25:37"></sun></city><temperature value="47.82" min="43"
max="53.6" unit="fahrenheit"></temperature><feels_like value="41.41"
unit="fahrenheit"></feels_like><humidity value="43" unit="%"></humidity><pressure
value="1030" unit="hPa"></pressure><wind><speed value="3.71" unit="mph" name="Light
breeze"></speed><gusts></gusts><direction value="32" code="NNE"
name="North-northeast"></direction></wind><clouds value="1" name="clear
sky"></clouds><visibility value="16093"></visibility><precipitation
mode="no"></precipitation><weather number="800" value="clear sky"
icon="01n"></weather><lastupdate value="2020-03-08T00:17:16"></lastupdate></current>
```

or after formatting:

```
<?xml version="1.0" encoding="UTF-8"?>
<current>
    <city id="4575352" name="Columbia">
        <coord lon="-81.03" lat="34">
        </coord>
        <country>US</country>
        <timezone>-18000</timezone>
        <sun rise="2020-03-07T11:44:40" set="2020-03-07T23:25:37">
        </sun>
    </city>
    <temperature value="47.82" min="43" max="53.6" unit="fahrenheit">
    </temperature>
    <feels_like value="41.41" unit="fahrenheit">
    </feels_like>
    <humidity value="43" unit="%"></humidity>
    <pressure value="1030" unit="hPa"></pressure>
    <wind>
        <speed value="3.71" unit="mph" name="Light breeze"></speed>
        <gusts></gusts>
        <direction value="32" code="NNE" name="North-northeast">
        </direction>
    </wind>
    <clouds value="1" name="clear sky"></clouds>
    <visibility value="16093"></visibility>
    <precipitation mode="no"></precipitation>
    <weather number="800" value="clear sky" icon="01n"></weather>
    <lastupdate value="2020-03-08T00:17:16"></lastupdate>
</current>
```

Write a Python program so that when you run it, it displays the above information (which should be extracted using regular expressions). For instance the above XML data will result in the following output:

```
City: Columbia
Country: US
Sun rise: 2020-03-07T11:44:40
Sun set: 2020-03-07T23:25:37
Temperature: 47.82, min 43, max 53.6 fahrenheit
Feels like: 41.41 fahrenheit
Humidity: 43%
Pressure: 1030 hPa
Wind speed: 3.71 mph, Light breeze
Wind direction: North-northeast
Clouds: clear sky
Visibility: 16093
Precipitation: no
Last update: 2020-03-08T00:17:16
```

Of course if you run the program at different times, you will get different results.

Here's a skeleton:

```
# Name: John Doe
# File: main.py

import urllib2
import re

city_id = "4575352"
api_key = "014f77b0g63693crffcbhab9dfvb7903"
url = "http://api.openweathermap.org/data/2.5/weather?id=%s&mode=xml&units=imperial&APPID=%s"
url = url % (city_id, api_key)
xml = urllib2.urlopen(url).read()

city = ""
country = ""
sun_rise = ""

print "City:", city
print "Country:", country
print "Sun rise:", sun_rise
```

Your goal is to, of course, extract the city from the string xml using a regular expression:

```
# Name: John Doe
# File: main.py

import urllib2
import re

city_id = "4575352"
api_key = "014f77b0g63693crffcbhab9dfvb7903"
url = "http://api.openweathermap.org/data/2.5/weather?id=%s&mode=xml&units=imperial&APPID=%s"
url = url % (city_id, api_key)
xml = urllib2.urlopen(url).read()

...
city = ...
country = ""
sun_rise = ""

print "City:", city
print "Country:", country
print "Sun rise:", sun_rise
```

Then you need to extract the country.

```
# Name: John Doe
# File: main.py

import urllib2
import re

city_id = "4575352"
api_key = "014f77b0g63693crffcbhab9dfvb7903"
url = "http://api.openweathermap.org/data/2.5/weather?id=%s&mode=xml&units=imperial&APPID=%s"
url = url % (city_id, api_key)
xml = urllib2.urlopen(url).read()

...
city = ...
...
country = ...
sun_rise = ""

print "City:", city
print "Country:", country
print "Sun rise:", sun_rise
```

Etc. (It's even better if you can write one single regex to extract all the data.)

Once your program is working, change it so that works for any citiy id:

```
# Name: John Doe
# File: main.py

import urllib2
import re

city_id = raw_input("enter city id: ")
api_key = "014f77b0g63693crffcbhab9dfvb7903"
url = "http://api.openweathermap.org/data/2.5/weather?id=%s&mode=xml&units=imperial&APPID=%s"
url = url % (city_id, api_key)
xml = urllib2.urlopen(url).read()

...
```