Name: Navraj Narula
Class: NLP - Problem Set 1
Date Due: February 12, 2016

1. Create regular expressions for each component of the date expression. Make sure that your RE are as succinct as possible by using the operators that we discussed in class. Explain in a sentence or two how you built each one of your RE.

1.1. MM component: Assume that months are in [01, 02, ..., 12] inclusive.
1.2. DD component: Assume that days are in [01, 02, ..., 31] inclusive, irrespective of which month/year it is. (This is a simplification)
1.3. YYYY component: Assume that years are in [1900, 1901, ..., 2099] inclusive.
1.4. Separator component: Assume that valid separators are space, dash and forward slash. You do not have to force the two separators in MM/DD/YYYY to match, that is, we will assume that 1231/2000and 12/312000are also valid dates.

SOLUTION (tested via http://regexr.com/):

* Note: Escape characters are included in the regex expression below.

(0[1-9]|1[0-2])(-|\/|)(0[1-9]|1[0-9]|2[0-9]|3[0-1])(-|\/|)((19|20)[0-9]{2})

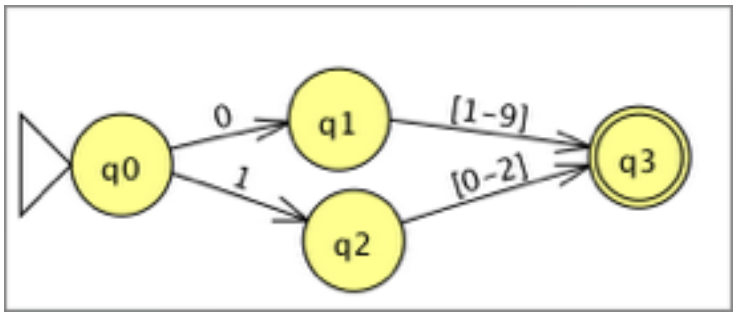
I constructed this RE by breaking down its components into four parts: MM, DD, YYYY, and the allowed separators (i.e. "/", "-", or " " which is a *blank space*). For MM, a disjunction allowed me to specify restrictions on the digit in the ones place depending on whether or not the digits in the tenths place was a "0" or "1." What must come after MM is a separator, which I also specified in terms on disjunction. DD was constructed much in the same as MM, except the digits differ. "0," "1," or "2" may take the tenths place followed by any digit. "3" may take the tenths place followed by either a "0" or "1." Separators again are anticipated afterwards. YYYY has been restricted to begin with a "19" or "20" and then is able to be followed by any two digits. All these parts must be present for the date string to return a match.

2. Draw FSA corresponding to each component and describe how you would build an FSA for the entire expression out of these individual components (No need to draw the final FSA as it might be too large). When you draw each FSA, include a paragraph explaining your design choices. Number your states with integers so that 0 is always the initial state. Clearly mark your initial and final states.

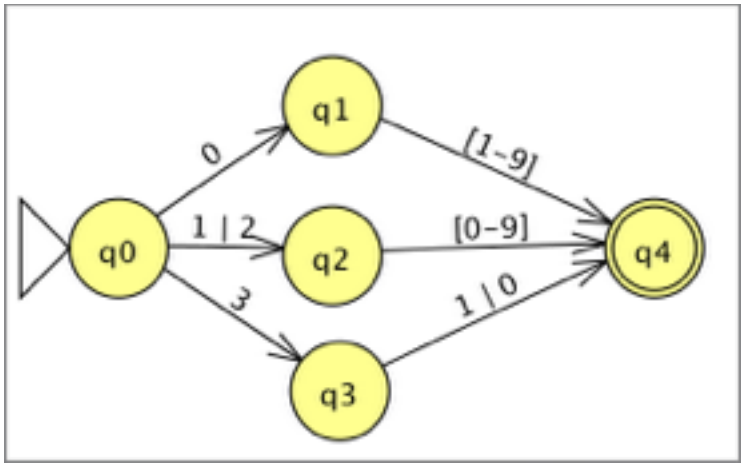
* Note: I used JFLAP to draw my FSAs.

SOLUTION:

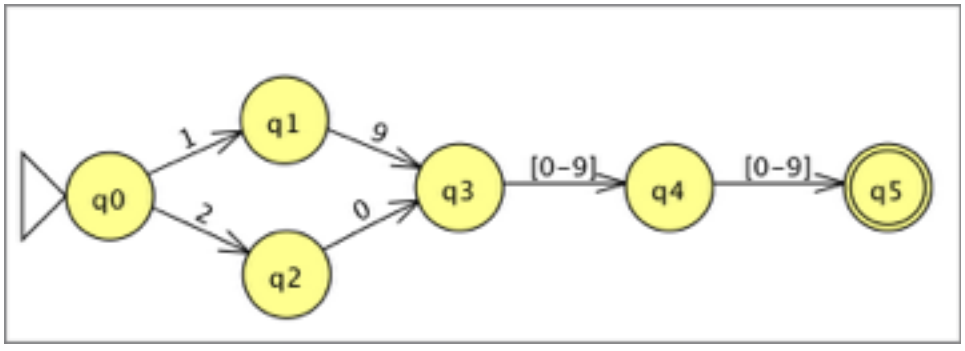
I will draw each component of my regular expression separately: MM, DD, YYYY, an separators. Then, I will explain how these separate FSAs can be concatenated together to return match for an entire reggae expression for a date.



FSA for MM

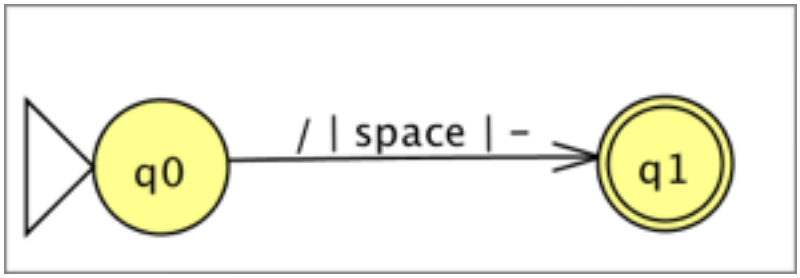


FSA for DD

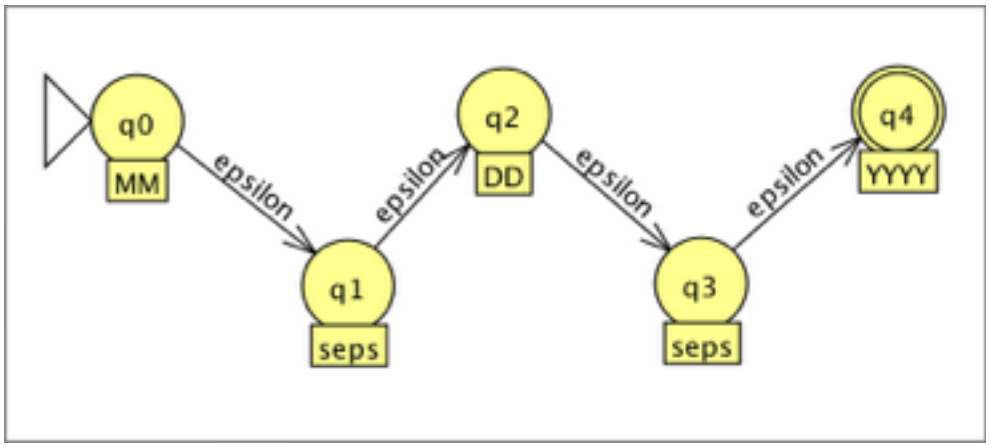


FSA for

YYYY



FSA for separators



The final FSA should resemble the diagram above. There is one initial state and one final state. Epsilon transitions connect the final state of each preceding FSA to the initial state of the anticipating FSA. This allows only valid date strings to be accepted and all others to be rejected.

3. Implement an FSA class in Python. In **pset1_template.py** we provide you with a recommended prototype that you can follow. You can use your own design if you prefer but make sure to explain your choices. Create FSA objects for each of the components above.

I constructed the FSA class according to the design that was given. Transitions to a new state were made in terms of a reassignment based on the observed input in the current state. Lookups were involved in ensuring the correctness of this by means of checking the tuple that contained the current input and the current state. Final states (i.e. accepting states were also designated within this FSA class. The specific function definitions I added are below.

```
def add_transition(self, s, currentState, newState):
    self.transitions[(s, currentState)] = newState

def set_final_state(self, final):
    self.final_states.add(final)

def lookup(self, current, s):
    if (s, current) in self.transitions:
        return self.transitions[(s, current)]
    else:
        return None

def is_final(self, state):
    return state in self.final_states
```

4. Implement the DRECOGNIZE algorithm of SLP Figure 2.12 using your FSA class. The signature of this function should be such that we can call it as `D Recognize("01", fsa)` a d i t r e t u r n s `True` o r `F alse`.

The DRecognize algorithm, as I understand it, will not take in an entire date string, but rather a chunk of it (such as '12' or '11' or '/' or '1993') along with one FSA (MM, seps, DD, YYY) and return either True or False. A crucial step in implementing this algorithm is checking to see whether or not a final state has been reached, but also knowing when to perform a "lookup" if the string being read has not yet—this was accomplished by calling on the lookup function defined in the FSA object, indicating the current state as well as the index of the string of the current part of the string being observed by the function.

```
Testing Months Example...
DRecognize(12, months)   True
DRecognize(13, months)   False
```

Example from running code

5. Extend DRECOGNIZE into D RecognizeMulti s o that it takes as input a list of FSA instead of a single one. This algorithm should accept/reject input strings such as *12/31/2000* based on whether or not the string is in the language defined by the FSA that is the concatenation of the input list of FSA. Use the Test function provided in pset1_template.py to demonstrate your algorithm on recognizing date expressions.

The DRecognizeMulti algorithm takes in an entire date string (i.e. '12/11/1993') and multiple FSAs (MM, seps, DDs, etc. in a list). My previous implementation for this function involved nested for loops; however, I was failing on several results such as: '123' or '//' due to a result of accepting to early. I finally tried another method, which involved recursion and this seemed to fixed the issue quicker than I could have using for loops. The base involved checking to whether or not a final state had been reached depending on the length of the input string. Else, I would continue checking the "rest" (i.e. DRecognizeMulti(input_str[index:], fsa_list[1:])). This would only happen if the length of the FSA list is greater than one, though.

```
Test Days FSA
''      False
'00'    False
'01'    True
'09'    True
'9'     False
'10'    True
'11'    True
'21'    True
'31'    True
'32'    False
'123'   False
```

Example from running code

6. Introduce nondeterminism in your MM and DD components so that dates such as *1 / 2/2000* as well as *0 1/02/2000* can both be recognized (singledigit and doubledigit months and days): Extend your FSA class so that it supports nondeterministic FSA. Implement NDRECOGNIZE of SLP Figure 2.19 and demonstrate that it accepts/rejects valid/invalid date expressions. You can do this in one of two ways:

. 6.1. Implement NDRecognize and N DRecognizeMulti (similar to steps 4 and 5).

. 6.2. Build one large nondeterministic automaton for date expressions MM/DD/YYYY and use NDRecognize.

I implemented my nonderministic finite state machine in a separate file (pset1_extracredit.py). The process of doing so was not simple, but manageable to try since I mostly re-used code from implementing my deterministic finite state machine. I remember during our first class Professor Yamangil spoke about how an NDFSA works much like a stack. This was the primary data structure I used to implement this algorithm along with pseudocode from our SLP text. When you run the file, you will observe that it only works on date chunks (i.e. “1”, “2”, or “1993”, but no “1/2/1993.)

```
Test Days FSA
''      False
'00'    False
'1'     True
'3'     True
'9'     True
'10'    True
'11'    True
'21'    True
'31'    True
'32'    False
'123'   False
```

```
Test Date Expressions FSA
''      False
'2 31 2000'     False
'3/3/2000'      False
'2-31-2000'     False
'12:31:2000'    False
'00-31-2000'    False
'12-00-2000'    False
'12-31-0000'    False
'12-32-1987'    False
'13-31-1987'    False
'12-31-2150'    False
```