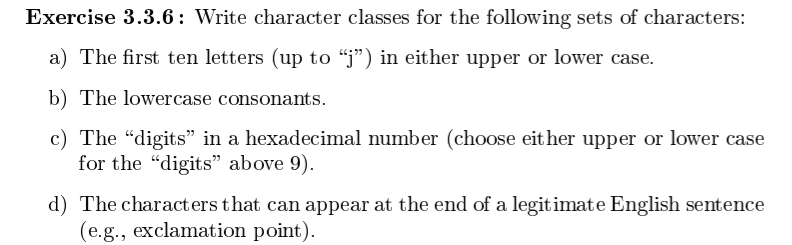
3.3.6,



A, [A-Ja-j]

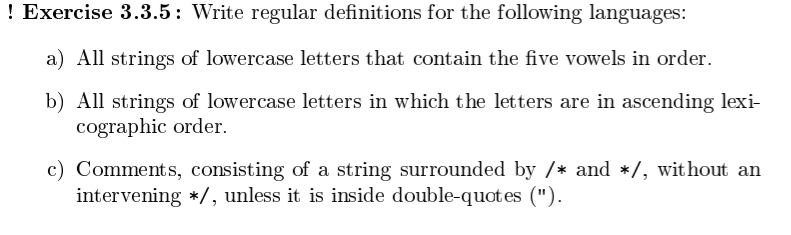
B, [bcdfghjklmnpqrstuwxyz] (Counting ‘y’ as a consonant.)

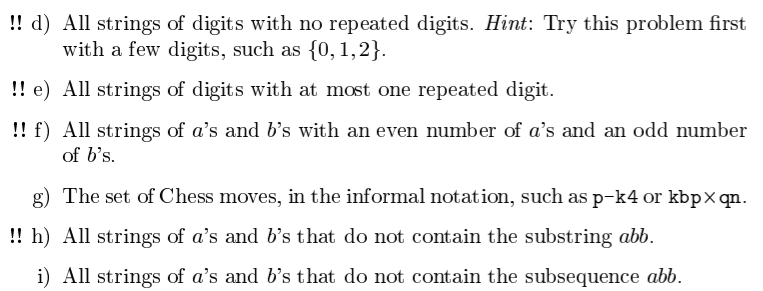
C, [0-9A-F]

D, [!.?]

3.4.2 provide DFA instead of transition diagram







DFA will be given via transition table form accepted by the below python function:

def finite\_automaton(input, transition\_table, accepting\_states):

current\_state = 0

i = 0

while (i < len(input)):

current\_state = transition\_table[current\_state][input[i]]

i += 1

if (current\_state in accepting\_states):

return True

else:

return False

Sample DFA: {0:{'a':1, 'b':0}, 1:{'a':1, 'b':2}, 2:{'a':1, 'b':3}, 3:{'a':1, 'b':0}}

As an example, the first state in the above reads: For state ‘0’, the input ‘a’ goes to state 1, and the input ‘b’ goes to state 0.

Sample Accepting State(s): State 3

(Above DFA accepts strings of a’s and b’s ending in “abb”)

A, All strings of lower case letters that contain the five vowels (aeiou) in order.

Assuming that [a-z] are the only possible inputs. (Otherwise state 0 should have an extra transition state that goes from [^a-z] to a rejecting state that goes to itself on all inputs.) Alternatively, it can be assumed that inputs that don’t go to a state automatically cause a rejection of the entire string.

Note that I’ll be using the character class shorthand although the python program won’t accept them. (That is, {[a-c]: 3} is equivalent to {a:3, b:3, c:3}.)

Transition Table:

{0:{[^a]:0, a:1},

1:{a:1, e:2, [^ae]:0},

2:{a:1, i:3, [^ai]:0},

3:{a:1, o:4, [^ao]:0},

4:{a:1, u:5, [^au]:0},

5:{[a-z]:5}}

Accepting State(s): [5] ([a,b,c] means states a, b, and c are accepted, so [5] means state 5 is accepted.)

B, All strings of lowercase letters in which the letters are in ascending lexicographic order.

(Assuming ascending order means the list [b, a, c] gets sorted into [a, b, c].)

Assuming that the empty string is a string of all lowercase letters, and that, additionally, the empty string and a string with only one letter is sorted. (That is, the DFA accepts the empty string and strings of a single lowercase character.) Also assuming that each letter can appear more than once, that is, “aabbcc” would be accepted.

Transition Table:

{0:{a:a, b:b, c:c, d:d, e:e, f:f, g:g, h:h, i:i, j:j, k:k, l:l, m:m, n:n, o:o, p:p, q:q, r:r, s:s, t:t, u:u, v:v, w:w, x:x, y:y, z:z},

1:{[a-z]:1},

a:{a:a, b:b, c:c, d:d, e:e, f:f, g:g, h:h, i:i, j:j, k:k, l:l, m:m, n:n, o:o, p:p, q:q, r:r, s:s, t:t, u:u, v:v, w:w, x:x, y:y, z:z},

b:{[a]: 1, b:b, c:c, d:d, e:e, f:f, g:g, h:h, i:i, j:j, k:k, l:l, m:m, n:n, o:o, p:p, q:q, r:r, s:s, t:t, u:u, v:v, w:w, x:x, y:y, z:z},

c:{[a-b]: 1, c:c, d:d, e:e, f:f, g:g, h:h, i:i, j:j, k:k, l:l, m:m, n:n, o:o, p:p, q:q, r:r, s:s, t:t, u:u, v:v, w:w, x:x, y:y, z:z},

d:{[a-c]: 1, d:d, e:e, f:f, g:g, h:h, i:i, j:j, k:k, l:l, m:m, n:n, o:o, p:p, q:q, r:r, s:s, t:t, u:u, v:v, w:w, x:x, y:y, z:z},

e:{[a-d]: 1, e:e, f:f, g:g, h:h, i:i, j:j, k:k, l:l, m:m, n:n, o:o, p:p, q:q, r:r, s:s, t:t, u:u, v:v, w:w, x:x, y:y, z:z},

f:{[a-e]: 1, f:f, g:g, h:h, i:i, j:j, k:k, l:l, m:m, n:n, o:o, p:p, q:q, r:r, s:s, t:t, u:u, v:v, w:w, x:x, y:y, z:z},

and so on until

z: {[a-y]:1, z:z}

}

Basically, each state transitions to the state named by the input unless the input comes before the state, in which case we transition to the reject state. (If we assume transitions that aren’t listed immediately cause a reject we can also simply not list the inputs that come before the state in alphabetic order.)

Accepting States: [0, [a-z]], that is, all states except state 1. (Alternatively, the states 0 and all states from a to z.)

If the empty string should not be accepted remove the empty string from the list of accepting states.

C, Comments, consisting of a string surrounded by /\*/ and \*/, without an intervening \*/, unless it is inside double-quotes (“).

Uncertain if that means /\*/ “” \*/ “” \*/ should be accepted. (The intervening \*/ is between two double quotes, but obviously not inside any pair of double quotes.) For example, should /\*/”\*/”\*/”\*/”\*/ be accepted? And, would /\*/”\*/\*/\*/”\*/”\*/ be accepted?

Additionally, not sure if only “\*/” should be allowed or if “abc\*/” and the like are also allowed to be between /\*/ and \*/.

I’m going to use an extremely permissive definition, that is, anything between two quotes will be allowed, even if there are quotes inside quotes. E.x. /\*/””””””””””””\*/ is equivalent to /\*/””\*/ for the purposes of this DFA---if there are more than two double quotes in the string then only the first and last double quote matters for determining what is ‘inside.’

We can treat “/\*/” and “\*/” as single tokens, distinct from “/” and “\*” individually, to save states.

The token ‘any’ stands for any and all input.

The tokens of the form any[^abc] means any and all input EXCEPT a, b, or c. Similarly, any[^a-z] would be any input except lowercase letters between a and z, inclusive.

Transition table:

{0:{/\*/:1, any[^/\*/]:reject} #have to start with /\*/

1:{\*/:no\_quote\_accept, “:seen\_quote, any[^\*/”]: 1},

no\_quote\_accept:{any:reject}, #We haven’t seen a quote, so if \*/ didn’t end the input we reject

reject:{any:reject},

seen\_quote:{\*/:seen\_quote\_accept, any[^\*/]:seen\_quote},

seen\_quote\_accept: {any[^”]:seen\_quote\_and\_\*/, “:seen\_quote}, #We *have* seen a quote, so it’s possible this input still works, however, we would need to see a quote before seeing \*/ as the last symbol to accept.

Seen\_quote\_and\*/:{ any[^”]:seen\_quote\_and\_\*/, “:seen\_quote} #Note that we move back to seen\_quote after seeing “ rather than 1 because we’re using a permissive definition

}

Accepting states = [no\_quote\_accept, seen\_quote\_accept]

D, All strings of digits with no repeated digits. Each state must remember all digits seen before, so in base 10 there would be roughly 2^10 states, or around a thousand states.

I’m not going to write that out, so let’s use a base 4 system which is complicated enough to showcase how the transition table should be built, and simple enough to be built by hand and human readable.

That is, inputs are of the form [0-3]. Any inputs not specified leads to rejection. (This means we don’t have to specifically reject even inputs in [0-3] if we already seen it before.)

Assuming that the empty string is accepted.

{false\_false\_false\_false:{0:true\_false\_false\_false, 1:false\_true\_false\_false, 2:false\_false\_true\_false, 3:false\_false\_false\_true}, #states are named by the digits we’ve seen

true\_false\_false\_false:{1:true\_true\_false\_false, 2:true\_false\_true\_false, 3:true\_false\_false\_true},

false\_true\_false\_false:{0:true\_true\_false\_false, 2:false\_true\_true\_false, 3:false\_true\_false\_true},

false\_false\_true\_false:{0:true\_false\_true\_false, 1:false\_true\_ true\_false, 3:false\_false\_true\_true},

false\_false\_false\_true:{0:true\_false\_false\_true, 1:false\_true\_false\_true, 2:false\_false\_true\_true},

true\_true\_false\_false:{2:true\_true\_true\_false, 3: true\_true\_false\_true},

true\_false\_true\_false:{1:true\_true\_true\_false, 3: true\_false\_true\_true},

true\_false\_false\_true:{1:true\_true\_false\_true, 2: true\_false\_true\_true},

false\_true\_true\_false:{0:true\_true\_true\_false, 3: false\_true\_true\_true},

false\_true\_false\_true:{0:true\_true\_false\_true, 2:false\_true\_true\_true},

false\_false\_true\_true:{0:true\_false\_true\_true, 1:false\_true\_true\_true},

true\_true\_true\_false:{3:true\_true\_true\_true},

true\_true\_false\_true:{2:true\_true\_true\_true},

true\_false\_true\_true:{1:true\_true\_true\_true},

false\_true\_true\_true:{0:true\_true\_true\_true},

true\_true\_true\_true:{} #no transitions, any input leads to rejection

}

Accepting States: [any] That is, all of them. (We’re using the shorthand of transitions not listed automatically lead to a permanently rejecting state, but which is not listed in the list of states.)

E, All strings of digits with at most 1 repeated digit.

Base 4 would be too complicated for this because we now need to store an additional bit of information, whether we’ve already seen a repeat or not, so the number of states for a base 4 solution to this would be equivalent to a base 5 solution of the previous problem.

In addition, I’m getting sick of writing out all the states at this point, so we’ll use base 2 rather than base 3, like you’d expect.

Transition table:

{false\_false\_false:{0:true\_false\_false, 1:false\_true\_false}, #three Booleans but only 2 possible inputs, the third Boolean stores whether we’ve seen a repeat or not

true\_false\_false:{0:true\_false\_true, 1:true\_true\_false},

false\_true\_false:{0:true\_true\_false, 1:false\_true\_true},

true\_true\_false:{[01]:true\_true\_true}, #[01] means 0 or 1

true\_false\_true:{1:true\_true\_true},

false\_true\_true:{0:true\_true\_true},

true\_true\_true:{}, #no transitions, all input lead to rejection

}

Accepting States: [any], that is, all of the listed states are accepted, under the assumption that the empty string is accepted and that transitions that aren’t listed automatically go to a permanent rejection state, which also isn’t listed.

F, All strings of a’s and b’s with an even number of a’s and an odd number of b’s.

Assuming that 0 is even.

Transition Table:

{even\_even:{a:odd\_even, b:even\_odd},

odd\_even:{a:even\_even, b:odd\_odd},

even\_odd:{a:odd\_odd, b:even\_even},

odd\_odd:{a:even\_odd, b:odd\_even}

}

Accepting States: [even\_odd] The only accepting state is when we’ve seen an even number of a’s and an odd number of b’s.

G, The set of chess moves in the information notation, such as p-k4 or kbpxqn.

Using the algebraic notation rules in : <https://web.archive.org/web/20131003050537/http://www.fide.com/component/handbook/?id=125&view=article>, with the exception that P stands for pawn rather than making the assumption that unnamed pieces are pawns.

Dictionary of tokens:

Piece = [KQRBNP] (King, Queen, Rook, Bishop, Knight, Pawn)

File = [a-h]

Rank = [1-8]

Regular movement has the form: PieceFileRank

Capture has the form: Piece’x’FileRank

Pawn Capture has the form: PieceFile’x’FileRank with optional “ e.p.” at the end to signify en passant (I’m not implementing en passant.)

If two identical pieces can move to the same square:

PieceFileFileRank

Or

PieceRankFileRank

If the identical pieces can capture:

PieceFile’x’FileRank

Or

PieceRank’x’FileRank

If two pawns can capture the form is:

PieceFile’x’FileRank (because I treat pawns identical to other pieces, we already have this covered under identical piece captures)

Pawn promotion is:

PieceFileRankPiece

I’m ignoring things like en passent, castling, check, checkmate, or draws.

If

Transition Table:

{0: {Piece: 1}, #has to start with a piece since pawns are P in my notation

1:{File: 2, Rank: 3, x: Capture},

2:{x:Capture, Rank: Pawn\_Promotion, File: Move\_Rank}

3:{x:Capture, File: Move\_Rank}

Capture{File:Capture\_Rank}, #All captures end with FileRank

Capture\_Rank:{Rank: Accept},

Move\_Rank: {Rank: Accept},

Pawn\_Promotion: {Piece: Accept} #it’s possible that this is accept, but also possible it’s a pawn promotion

Accept{}

}

Accepting States: [Accept, Pawn\_Promotion]

H, All strings of a’s and b’s that do not contain the substring abb.

Assuming that the empty string is accepted.

Transition Table:

{0: {a:a, b:0},

a:{b:ab, a:a},

ab:{a:a} #no transition for b because we reject if we see b here

}

Accepting States: [any], we accept 0, a, and ab because in those states we have yet to see abb, and we immediately reject if we see abb.

I, All strings of a’s and b’s that do not contain the subsequence abb. (This means once we see an a we’re no longer allowed to see 2 b’s, ever.)

Again assuming the empty string is accepted.

Transition Table:

{0: {a:a, b:0},

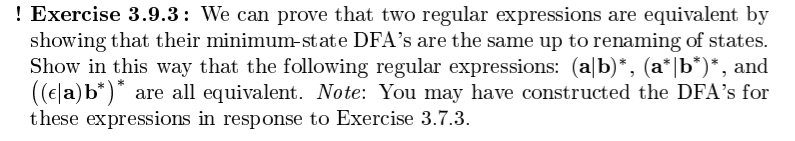
a:{a:a, b:ab},

ab:{a:ab} #go to ab on a because we need to keep remembering we already saw a b, so the next one leads to rejection.

}

Accepting States: [any], accept on 0, a, and ab because in those states we haven’t seen an a followed by 2 b’s yet, and we’ll automatically reject if we do.

3.9.3,



We need to build DFAs for the Res and then reduce to prove equivalence.

Unfortunately we’ll need to follow the production rules of DFAs then reduce rather than just building the DFAs directly (which would result in the identical {0: {[ab]:0}} transition table and 0 being accepting DFA for all three REs).

(a|b)\*:

First, a|b creates the state:

0:{a:1, b:1} #seeing a or b advances us to state 1 from state 0. (Simplified, we should technically give state 0 null transitions to state 1 and 2, state 1 advancing to state 3 on a, and state 2 advancing to 3 on b.)

Then, the \* means that state 1 has a null transition to state 0, and state 0 a null transition to state 1. With state 1 being the accepting state.

e = epsilon

Transition Table:

{0{a:1, b:1, e: 1},

1{e:0}

}

Accepting State(s): [1]

However, this is a NFA. Converting it to a DFA yields:

{0 :{[ab]:0}}

Accepting State(s): [0]

That is, we start at 0, go to 0 on a or b, and 0 is the accepting state. Since it’s not possible to distinguish a single state from itself, this DFA is minimized.

(a\*|b\*)\*:

NFA for a\*:

{0:{a:1, e:1}

1:{e:0}

}

Accept =[1]

NFA for b\*:

{0:{b:1, e:1}

1:{e:0}

}

Accept = [1]

NFA for a\*|b\*:

{0:{e:a0, e:b0},

a0:{a:a1, e:a1},

a1:{e:a0, e:1},

b0:{b:b1, e:b1},

b1:{e:b0, e:1}

1:{} #goes nowhere

}

Accept = [1]

Note that because the epsilon transitions go to a0 and b0 respectively, this does *not* simplify to the DFA for (a|b)\*.

NFA for (a\*|b\*)\*:

{0:{e:a0, e:b0, e:1},

a0:{a:a1, e:a1},

a1:{e:a0, e:1},

b0:{b:b1, e:b1},

b1:{e:b0, e:1}

1:{e:0}

}

Accept = [1]

Changing this to a DFA yields:

{0:{a:a, b:b},

a:{a:a, b:b},

b:{a:a, b:b},

}

And accept = [0, a, b]

None of the 3 states are distinguishable from each other (all accepting, and all 3 states respond to inputs the same way), so the DFA simplifies to:

{0:{[ab]:0}}

Identical to (a|b)\*

(a\*|b\*)\* and (a|b)\* are therefore equivalent.

((e|a)|b\*)\*:

Transition Table:

e|a:

{0:{e:1, a:1},

1:{}

}

Accept = [1]

(Technically 0 should have e transitions to e0 and a0, and e0 a e transition to e1, which has a e transition to 1, while a0 has an a transition to a1, which has an e transition to 1, but I omitted them for brevity.)

b\*:

Transition Table:

{0:{e:1, e:b0},

b0:{b:1},

1:{e:0}

}

Accept = [1]

(e|a)|b\*:

Transition Table:

{0:{e:e0, e:b0},

e0:{e:e1, a:e1},

e1:{e:1},

b0:{e:b1, e:bb1}

bb1:{b:b1}

b1:{e:b0, e:1}

1:{}

}

Accept = [1]

Converting (e|a)|b\* to a DFA yields the Transition Table:

{0:{a:a, b:b},

a:{},

b:{b:b}

}

Accept = [0, a, b]

((e|a)|b\*)\*:

Using the DFA for (e|a)|b\*, except we convert it to a NFA to allow closure.

Transition Table:

{0:{e:e0},

e0:{a:a, b:b, e:0},

a:{e:0},

b:{b:b, e:0}

}

Accept = [0]

Former accepting states have a null transition to the new starting state, allowing closure.

Cahnging the NFA to a DFA gives the Transition Table:

{0:{a:a, b:b},

a:{a:a, b:b},

b:{a:a, b:b}

}

Accept = [0, a, b]

This is identical to the unsimplified form of (a\*|b\*)\*. Consequently, reducing it to the minimum state will result in:

{0:{[ab]:0}

Accept = [0]

All simplified forms are identical, so all three regular expressions are identical.