CS/ECE 374 ✧ Fall 2019 / Section B

Homework 1

Due Tuesday, September 10, 2019 at 8pm

- This homework can be solved in a group of up to *three* students. Make sure that all three names and NetIDs appear on the paper, and that you mark all the group members in Gradescope

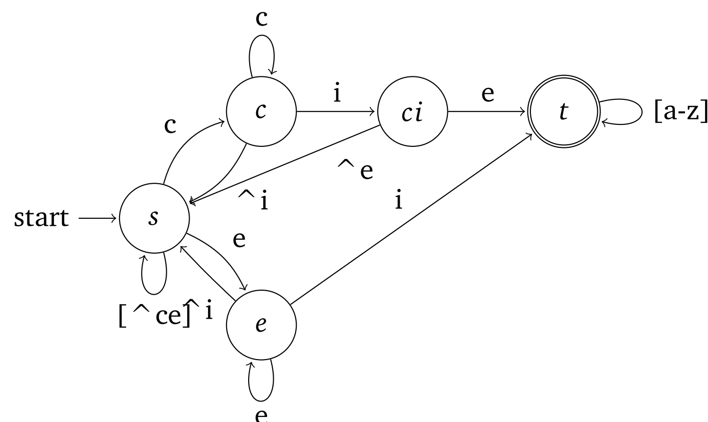
1. **Rules are meant to be broken.** An English spelling rule that is sometimes taught in school is: "*i* before *e* except after *c*." It can be used to remember what order to put the *e* and *i* in words such as *believe* or *receive*. But it turns out that this rule is also frequently wrong!

   (a) Create a regular expression over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}\}$ that describes strings that violate this principle. The corresponding language should include the words *feign* (*e* before *i*) and *concierge* (*i* before *e* after *c*), but not the words *tie*, *conceit* (rule is followed), *banana*, or *reminder* (no instance of *ei* or *ie*. You may use the shorthand $[a-d]$ to denote a range of symbols (i.e., $\texttt{a}+\texttt{b}+\texttt{c}+\texttt{d}$) and $[\hat{}e]$ to denote all symbols except $\texttt{e}$.

   Solution: `(ei+[a-z]*(cie+[^c]ei))[a-z]*`

   (b) Construct a DFA that recognizes the language above. You may use either a graphical representation or a formal listing of the states, transitions, and the starting and accepting states. In either case, describe what each of your states means and argue why you capture all the cases.

   Solution:



Now we define the DFA.

$M = (\Sigma, Q, s, A, \delta)$

$\Sigma = \{a, b, ..., z\}$

$Q = \{s, c, ci, e, t\}$

$s = s$

$A = \{t\}$

$$\delta(q,a) = \begin{cases} \delta(s, \hat{}[ce]) = s \\ \delta(s,c) = c \\ \delta(s,e) = e \\ \delta(c,c) = c \\ \delta(c, \hat{}i) = s \\ \delta(c,i) = ci \\ \delta(e,e) = e \\ \delta(e, \hat{}i) = s \\ \delta(e,i) = t \\ \delta(ci, \hat{}e) = s \\ \delta(ci,e) = t \\ \delta(t,[a-z]) = t \end{cases}$$

s: Start state and we have not just read e or c.
c: We just read a c.
e: We just read an e without substring ci immediately prior.
ci: We just read a i, giving us current substring ci.
t: We found substring cie or ei which breaks the rule.

(c) Construct a DFA that recognizes words that *both* break the rules and follow it, within the same word! You are looking for words such as *tietei* (not actually a word) that has the *i*–*e* pair in both the right and the wrong order at different points in the word. You may either use the product construction between the DFA in the previous part and another one, or create a new DFA directly. In both cases, explain what your states are and argue briefly why your DFA is correct.

Solution:

Define $M2$ that accepts strings that have and follow the rule.
$M2 = (\Sigma_2, Q_2, s_2, A_2, \delta_2)$
$\Sigma_2 = \{a, b, ..., z\}$
$Q_2 = \{s_2, c_2, ce_2, i_2, t_2\}$
$s = s_2$
$A = \{t_2\}$

$$\delta(q,a) = \begin{cases} \delta_2(s_2, \hat{}[ci]) = s_2 \\ \delta_2(s_2, c) = c_2 \\ \delta_2(s_2, i) = i_2 \\ \delta_2(c_2, c) = c_2 \\ \delta_2(c_2, \hat{}e) = s_2 \\ \delta_2(c_2, e) = ce_2 \\ \delta_2(i_2, i) = i_2 \\ \delta_2(i_2, \hat{}e) = s_2 \\ \delta_2(i_2, e) = t_2 \\ \delta_2(ce_2, \hat{}i) = s \\ \delta_2(ce, i) = t_2 \\ \delta_2(t_2, [a-z]) = t_2 \end{cases}$$

s_2: Start state and we have not just read c or i.
c_2: We just read a c.
i_2: We just read an i without substring ce immediately prior..
ce_2: We just read an e, giving substring ce.
t_2: We found substring cei or ie which follows the rule.

Let the DFA from 1.b. be M_1 Now do product construction for $M' = M_1 \times M_2$

$M' = (\Sigma', Q', s', A', \delta')$
$\Sigma' = \{a, b, ..., z\}$
$Q' = Q_1 \times Q_2$
$s' = (s_1, s_2)$
$A' = \{(t_1, t_2)\}$
$\delta'((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$

The states in the combined DFA are keeping track of the current substring, and once the rule is simultaneously broken and followed in the same word, we accept the state (t_1, t_2) since this indicates both types of substring were found.

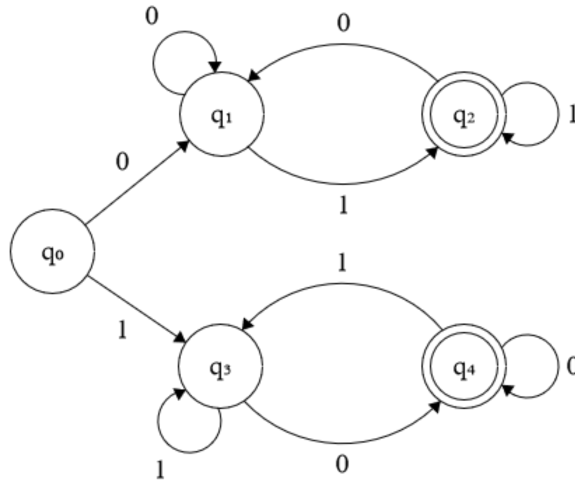
(d) (Not for submission) Create a regular expression that specifies the language from the previous part. Use it to find words in a dictionary (often stored in /usr/share/dict/words on Unix-like systems, e.g., MacOS or the EWS Linux machines) in this language.

Solution:

```
([a-z])*(cie + ^cei)([a-z])*(cei + ^cie)([a-z])* + ([a-z])*(cei
+ ^cie)([a-z])*(cie + ^cei)([a-z])* + ([a-z])*(iei + eie)([a-
z])*
```

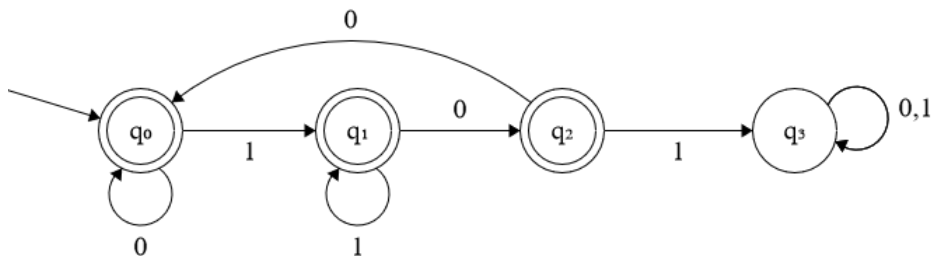
2. For each language described below, over the alphabet $\{0,1\}$, write (1) a regular expression for the language, and (2) a DFA for the language. Remember to explain the states in your DFA.

   (a) All strings that start and end with a different symbol
   Solution: `1(0+1)*0 + 0(0+1)*1`



q_0: We have not read anything yet.
q_1: The first character in the string was a `0`, and we just read a `0`.
q_2: The first character in the string was a `0`, and we just read a `1`.
q_3: The first character in the string was a `1`, and we just read a `1`.
q_4: The first character in the string was a `1`, and we just read a `0`.

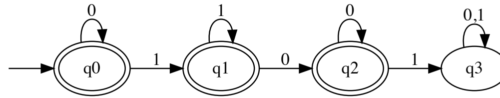
   (b) All strings that do not contain the substring 101
   Solution: `(0*11*00)*(0*+0*11*+0*11*0)`



q_0: Neither of the last two characters read was a `1`.
q_1: The last character read was `1` (but we as yet have not read `101`).
q_2: We just read the substring `10`.
q_3: We have read the substring `101`.

   (c) All strings that do not contain the *subsequence* 101. E.g., the string **1**111**0**000**1**111 would not be in the language because it contains the subsequence noted in bold.

Solution: $0^*1^*0^*$ (To prevent the subsequence 101 from occurring in the string, we must require that there be no 0s between any 1s—and the only way this will be true is if all 1s are adjacent to each other.)



q_0: We have read zero or more 0's so far.
q_1: We have read zero or more 0's followed by one or more 1's
q_2: We have read zero or more 0's followed by one or more 1's, followed by one or more 0's
q_3: We have found a subsequence 101

3. **Handcrafted artisanal DFAs.** Come up with three DFAs M_1, M_2, M_3 over the alphabet $\Sigma = \{0, 1\}$ that satisfy the following conditions:

   (a) $|Q_1| > 2$ and $|Q_2| > 2$

   (b) M_1 and M_2 use the *minimal* number of states for recognizing $L(M_1)$ and $L(M_2)$.

   (c) $L(M_1) \neq L(M_2)$

   (d) M_3 recognizes $L(M_1) \cap L(M_2)$

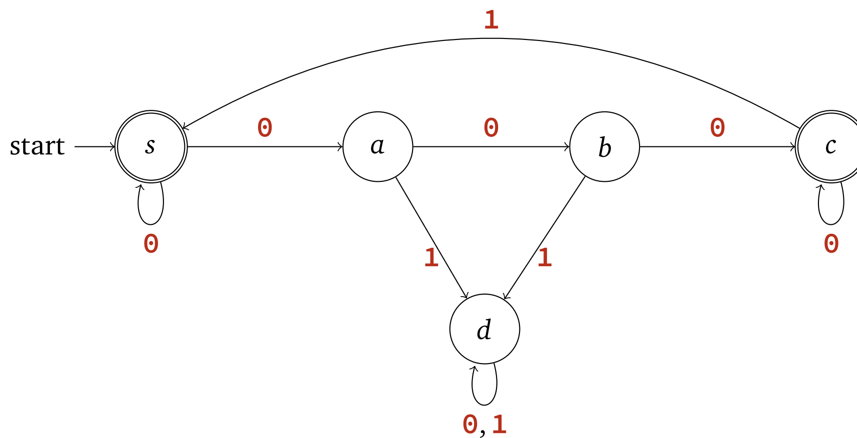
   (e) $|L(M_3)| = \infty$

   (f) $|Q_3| < |Q_1| \times |Q_2|$

   Your solution should give a description of each of the three languages, and an explanation of the states in each DFA. You also need to argue why your construction satisfies each of the points above.

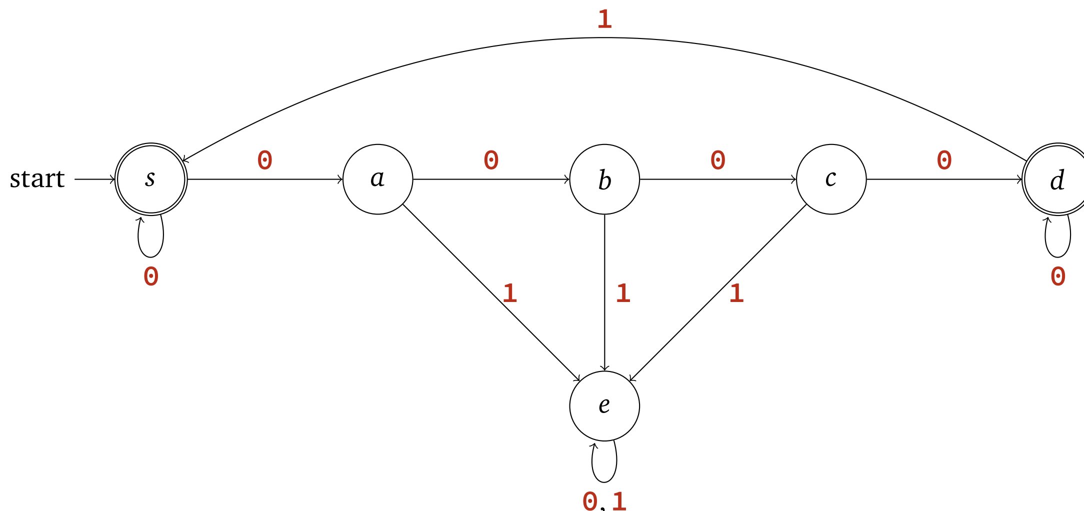
Solution (Not Unique):

M_1 accepts the strings in which every run of `0`s has length at least 3.



- s: We did not just read a `0`.
- a: We've read one `0` since the last `1` or the start of the string.
- b: We've read two `0`s since the last `1` or the start of the string.
- c: We've read at least three `0`s since the last `1` or the start of the string.
- d: We've read the substring `01` or `001`; reject.

M_2 accepts the strings in which every run of `0`s has length at least 4.



- s: We did not just read a 0.
- a: We've read one 0 since the last 1 or the start of the string.
- b: We've read two 0s since the last 1 or the start of the string.
- c: We've read three 0s since the last 1 or the start of the string.
- c: We've read at least four 0s since the last 1 or the start of the string.
- d: We've read the substring 01 or 001 or 0001; reject.

Obviously, (a) and (c) are satisfied; M_1 has 5 states, M_1 has 6 states, and $L(M_1) \neq L(M_2)$. Moreover, for M_1, 5 states are the minimum amount of states necessary to include all strings in which every run of 0s has length at least 3. s, a, b, c are used as counter for the number of consecutive 0s and e serves as a bad states that indicates less than 3 0s are detected in a single run. Similarly, M_2 is optimal when 6 states are used due to an extra counter state compared with M_1, so (b) is also satisfied.

In this setup, $M_1 \cap M_2 = M_2$, so let $M_3 = M_2$. Therefore, M_3 recognizes $L(M_1) \cap L(M_2)$ and (d) is satisfied. $L\{M_3\}$ denotes the language that contains all string in which every run of 0s has length at least 4 and there are infinite number of such finite strings, so $L\{M_3\} = \infty$ and (e) is satisfied.

$|Q_3| = |Q_2| = 6 < |Q_1| \times |Q_2| = 30$, so (f) is also satisfied. In conclusion, the constructions aboves for M_1, M_2, M_3 satisfied all six requirements.

■

Solved problem

4. ***C comments*** are the set of strings over alphabet $\Sigma = \{$⋆, /, A, ⋄, ↲$\}$ that form a proper comment in the C program language and its descendants, like C++ and Java. Here ↲ represents the newline character, ⋄ represents any other whitespace character (like the space and tab characters), and A represents any non-whitespace character other than ⋆ or /.[1] There are two types of C comments:

   - Line comments: Strings of the form //···↲.
   - Block comments: Strings of the form /⋆···⋆/.

   Following the C99 standard, we explicitly disallow ***nesting*** comments of the same type. A line comment starts with // and ends at the first ↲ after the opening //. A block comment starts with /⋆ and ends at the the first ⋆/ completely after the opening /⋆; in particular, every block comment has at least two ⋆s. For example, each of the following strings is a valid C comment:

   - /⋆⋆⋆/
   - //⋄//⋄↲
   - /⋆///⋄⋆⋄↲⋆⋆/
   - /⋆⋄//⋄↲⋄⋆/

   On the other hand, *none* of the following strings is a valid C comments:

   - /⋆/
   - //⋄//⋄↲⋄↲
   - /⋆⋄/⋆⋄⋆/⋄⋆/

   i. Describe a DFA that accepts the set of all C comments.

   ii. Describe a DFA that accepts the set of all strings composed entirely of blanks (⋄), newlines (↲), and C comments.

   You must explain *in English* how your DFAs work. Drawings or formal descriptions without English explanations will receive no credit, even if they are correct.

---

[1]The actual C commenting syntax is considerably more complex than described here, because of character and string literals.

- The opening /⋆ or // of a comment must not be inside a string literal (“···“) or a (multi-)character literal ('···').
- The opening double-quote of a string literal must not be inside a character literal ('"') or a comment.
- The closing double-quote of a string literal must not be escaped (\")
- The opening single-quote of a character literal must not be inside a string literal ("···'···") or a comment.
- The closing single-quote of a character literal must not be escaped (\')
- A backslash escapes the next symbol if and only if it is not itself escaped (\\) or inside a comment.

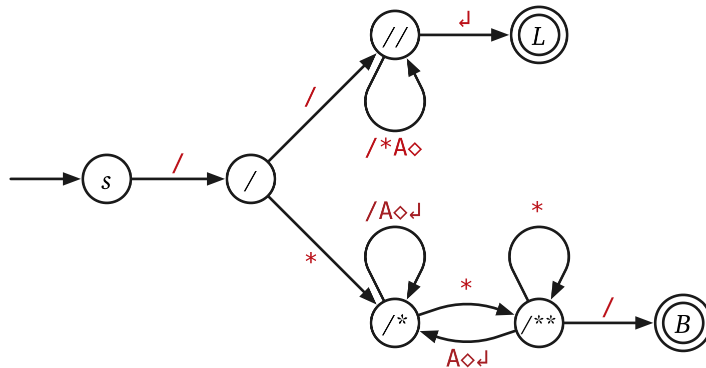
For example, the string "/⋆\\\"⋆/"/⋆"/⋆\"/⋆"⋆/ is a valid string literal (representing the 5-character string /⋆\"\⋆/, which is itself a valid block comment!) followed immediately by a valid block comment. ***For this homework question, just pretend that the characters ', ", and \ don't exist.***

Commenting in C++ is even more complicated, thanks to the addition of *raw* string literals. Don't ask.

Some C and C++ compilers do support nested block comments, in violation of the language specification. A few other languages, like OCaml, explicitly allow nesting block comments.

Solution:

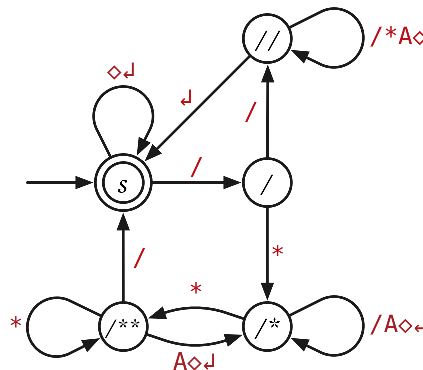
(a) The following eight-state DFA recognizes the language of C comments. All missing transitions lead to a hidden reject state.



The states are labeled mnemonically as follows:

- s — We have not read anything.
- / — We just read the initial /.
- // — We are reading a line comment.
- L — We have read a complete line comment.
- /* — We are reading a block comment, and we did not just read a * after the opening /*.
- /** — We are reading a block comment, and we just read a * after the opening /*.
- B — We have read a complete block comment.

(b) By merging the accepting states of the previous DFA with the start state and adding white-space transitions at the start state, we obtain the following six-state DFA. Again, all missing transitions lead to a hidden reject state.



The states are labeled mnemonically as follows:

- s — We are between comments.
- / — We just read the initial / of a comment.
- // — We are reading a line comment.
- /* — We are reading a block comment, and we did not just read a * after the opening /*.

- /** — We are reading a block comment, and we just read a * after the opening /*.

Rubric: 10 points = 5 for each part, using the standard DFA design rubric (scaled)

Rubric (DFA design): For problems worth 10 points:

- 2 points for an unambiguous description of a DFA, including the states set Q, the start state s, the accepting states A, and the transition function δ.
  - **For drawings:** Use an arrow from nowhere to indicate s, and doubled circles to indicate accepting states A. If $A = \varnothing$, say so explicitly. If your drawing omits a reject state, say so explicitly. **Draw neatly!** If we can't read your solution, we can't give you credit for it,.
  - **For text descriptions:** You can describe the transition function either using a 2d array, using mathematical notation, or using an algorithm.
  - **For product constructions:** You must give a complete description of the states and transition functions of the DFAs you are combining (as either drawings or text), together with the accepting states of the product DFA.
- **Homework only:** 4 points for *briefly* and correctly explaining the purpose of each state *in English*. This is how you justify that your DFA is correct.
  - For product constructions, explaining the states in the factor DFAs is enough.
  - **Deadly Sin:** ("Declare your variables.") No credit for the problem if the English description is missing, *even if the DFA is correct*.
- 4 points for correctness. (8 points on exams, with all penalties doubled)
  - −1 for a single mistake: a single misdirected transition, a single missing or extra accept state, rejecting exactly one string that should be accepted, or accepting exactly one string that should be accepted.
  - −2 for incorrectly accepting/rejecting more than one but a finite number of strings.
  - −4 for incorrectly accepting/rejecting an infinite number of strings.
- DFA drawings with too many states may be penalized. DFA drawings with *significantly* too many states may get no credit at all.
- Half credit for describing an NFA when the problem asks for a DFA.