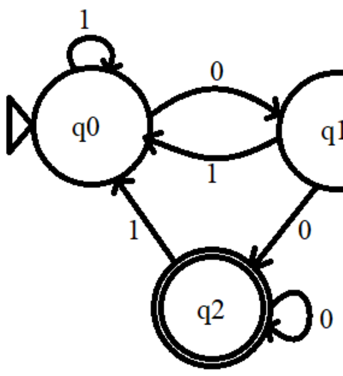
Math 590 Homework 2 Solutions Fall 2018

Due: Thursday, September 13

1. **Building NFAs**

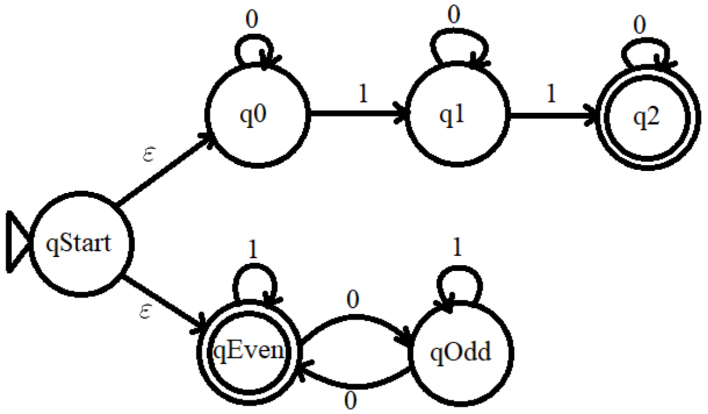
Draw NFAs for the following languages, with the specified number of states. You may assume the alphabet is always $\{0, 1\}$.

(a) The language $\{w \mid w \text{ ends in } 00\}$, 3 states.



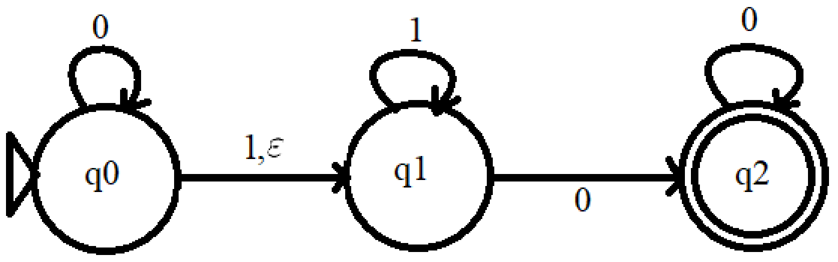
- $q_0$: just saw a 1
- $q_1$: just saw one 0
- $q_2$: just saw two 0s

(b) The language $\{w \mid w \text{ contains an even number of 0s or exactly two 1s}\}$, 6 states.



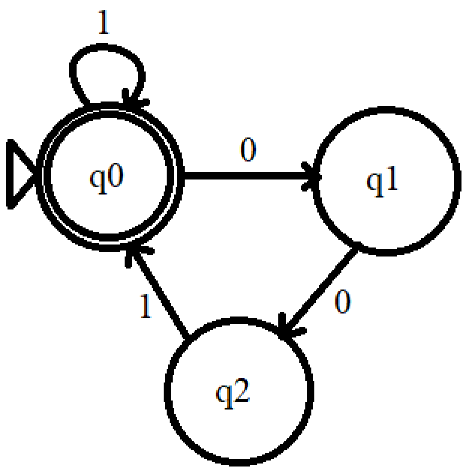
- $qStart$: start state for union
- $q_0$: zero 1s
- $q_1$: one 1
- $q_2$: two 1s
- $qEven$: even 0s
- $qOdd$: odd 0s

(c) The language $0^*1^*0^+$, 3 states.



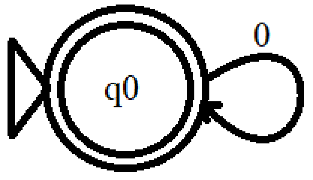
- $q_0$: $0^*$ (not accepting because concatenated)
- $q_1$: $1^*$ (not accepting because concatenated)
- $q_2$: $0^+$

(d) The language $1^*(001^+)^*$, 3 states.



- $q_0$: $1^*$ (also represents $1^+$ when reached from $q_2$)
- $q_1$: 0
- $q_2$: 0

(e) The language $0^*$, 1 state.

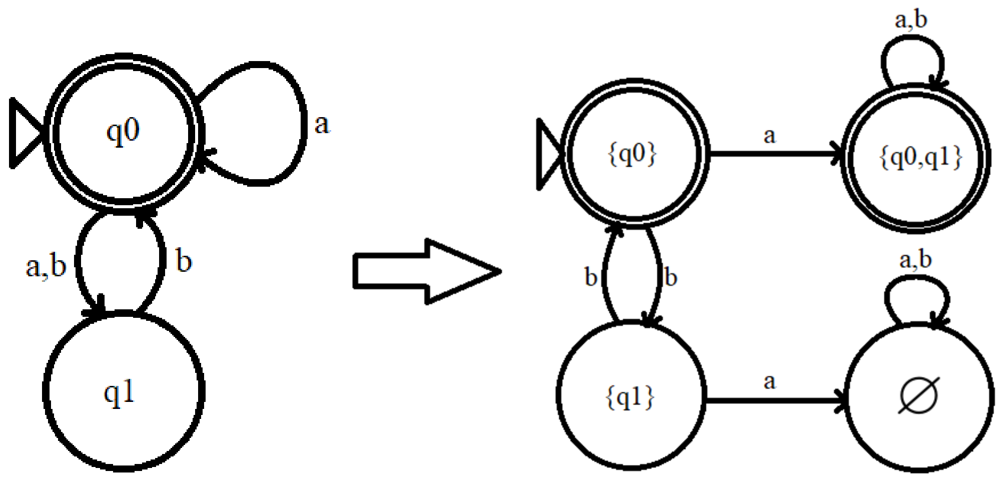


- $q_0$: $0^*$

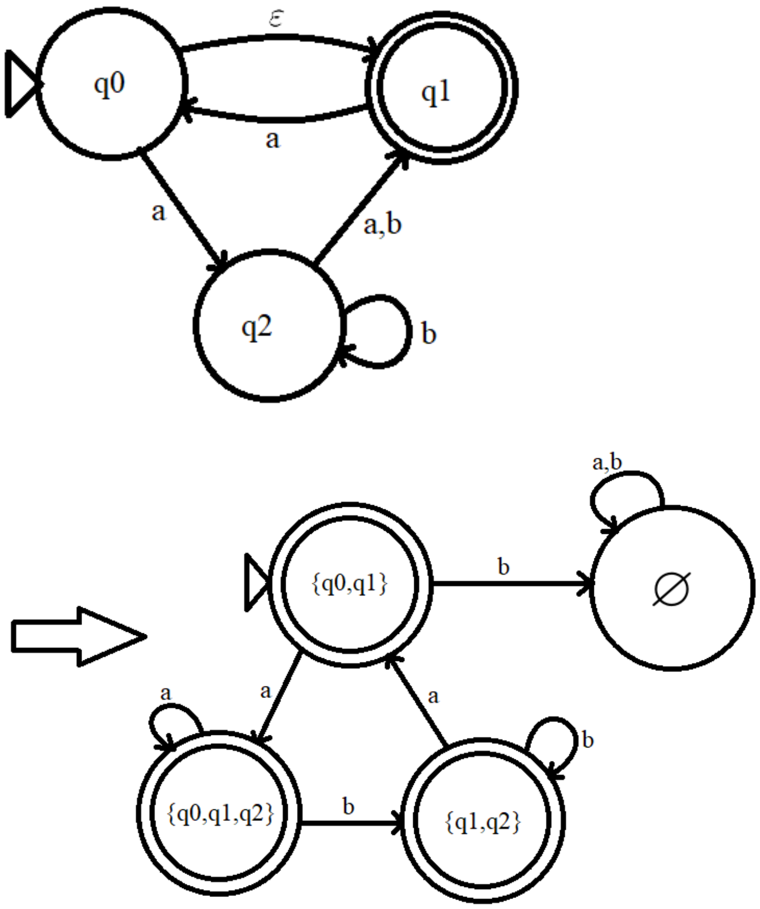
2. **Subset Construction**

Turn the following NFAs into DFAs using the subset construction. It should be clear from your drawing of a DFA how the subset construction was applied.

(a)



(b)



3. **Reversing a Language and Building a Binary Adder**

If $A$ is a language, we define its reverse, *reverse*$(A)$, as the language of all strings in $A$ written in reverse, i.e. *reverse*$(A) = \{w^{\mathcal{R}} \mid w \in A\}$. For example, if the string '00110' is in language $A$, then the string '01100' is in *reverse*$(A)$.

(a) Prove that the set of regular languages are closed under the reverse operation, i.e., prove that if language $A$ is regular, then *reverse*$(A)$ is also regular. (Hint: if $A$ is regular, then there is an NFA $M_1$ that recognizes it. How would you change that NFA into a new machine $M_2$ that recognizes the reverse?)

Take the machine $M_1$ that recognizes language $A$. Make its start state $q_0$ into an accepting state, and its accepting states $F$ into non-accepting states. Add a new starting state *qStart* and connect this state with $\varepsilon$ transitions to the formerly accepting states in $F$. Finally, reverse the direction of all of the remaining transitions in the machine. This new machine will start in *qStart* and only accept a string that can follow a path from a state in $F$ to the state $q_0$, i.e. a string that originally would have followed a path from $q_0$ to an accepting state in $F$.

(b) Define the alphabet

$$\Sigma_3 := \left\{ \begin{bmatrix}0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\end{bmatrix}, \begin{bmatrix}0\\1\\0\end{bmatrix}, \begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\1\end{bmatrix}, \begin{bmatrix}1\\0\\1\end{bmatrix}, \begin{bmatrix}1\\1\\0\end{bmatrix}, \begin{bmatrix}1\\1\\1\end{bmatrix} \right\}.$$

The alphabet $\Sigma_3$ contains eight size-3 columns of 0s and 1s. A string of symbols in $\Sigma_3$ thus builds threes rows of 0s and 1s. Consider each row to be a binary number and define the language

$$B = \{w \in \Sigma_3 \,|\, \text{the bottom row of } w \text{ is the sum of the top two rows}\}.$$

For example,

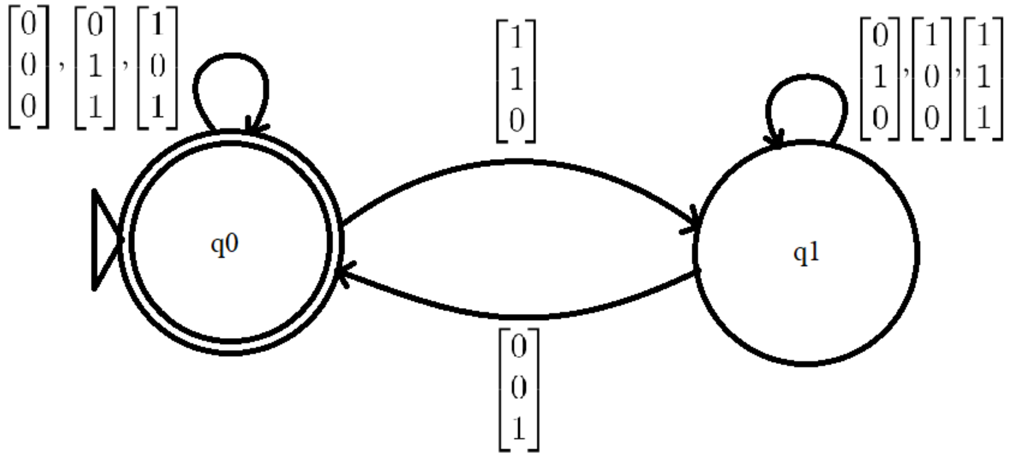
$$011 + 001 = 100 \quad \text{but} \quad 01 + 00 \neq 11,$$

so

$$\begin{bmatrix}0\\0\\1\end{bmatrix} \begin{bmatrix}1\\0\\0\end{bmatrix} \begin{bmatrix}1\\1\\0\end{bmatrix} \in B \quad \text{but} \quad \begin{bmatrix}0\\0\\1\end{bmatrix} \begin{bmatrix}1\\0\\1\end{bmatrix} \notin B.$$

Show that $B$ is regular. (Hint: show that *reverse*$(B)$ is regular, and by the previous problem conclude that *reverse*(*reverse*$(B)) = B$ is also regular.)

We will create a machine that recognizes *reverse*$(B)$, which will prove that *reverse*$(B)$ is a regular language. Then by problem 3(a), we know that $B$ must also be regular. We consider *reverse*$(B)$ so that we see the 1's place first. Our machine will have two states: $q_0$ is the 'carry 0' state, and $q_1$ is the 'carry 1' state. We get:



4. **Regular Expressions**

Give regular expressions that describe each of the following languages. You may assume that the alphabet in each case is $\Sigma = \{0, 1\}$.

(a) $\{w \mid \text{the length of } w \text{ is odd}\}$

$$(0|1)\ \ (\,(0|1)(0|1)\,)^*$$

(b) $\{w \mid w \text{ has an odd number of 0s}\}$

$$1^*0\ (\,01^*0\,|\,1\,)^*$$

(c) $\{w \mid w \text{ contains at least two 0s or exactly two 1s}\}$

$$(\ 1^*01^*0\,(0|1)^*\ \ |\ 0^*10^*10^*\ )$$

(d) $\{w \mid w \text{ does not end in a double letter, i.e., does not end in 00 or 11}\}$

$$(\ \varepsilon\,|\,0\,|\,1\,|\,(0|1)^*\,(01|10)\ )$$

(e) $\{w \mid w \text{ begins with a 1 and ends with a 0}\}$

$$1\,(0|1)^*\,0$$

(f) $\{w \mid \text{every odd position of } w \text{ is a 1}\}$

$$(\ (\,1(0|1)\,)^*\ \ |\ \ (\,1(0|1)\,)^*\,1\ )$$

5. **Lexers/Tokenizers**

   A *lexer* (also known as a *tokenizer*) is a program that takes a sequence of characters and splits it up into a sequence of words, or "tokens." Compilers typically do this as a prepass before parsing programs. For example, "`if (count == 42) ++n;`" might divide into `if` , `(` , `count` , `==` , `42` , `)` , `++` , `n` , `;` .

   Regular expressions are a convenient way to describe tokens (e.g., C integer constants) because they are unambiguous and compact. Further, they are easy for a computer to understand: *lexer generators* such as `lex` or `flex` can turn regular expressions into program code for dividing characters into tokens.

   In general, there may be many ways to divide the input up into tokens. For example, we might see the input `ifoundit = 1` as starting with a single token `ifoundit` (a variable name), or as starting with the keyword `if` followed immediately by the variable `oundit`. Most commonly, lexers are implemented to be *greedy*: given a choice, they prefer to produce the longest possible tokens. (Hence, `ifoundit` is preferred over `if` as the first token.)

   Lexers commonly skip over whitespace and comments. A "traditional" comment in C starts with the characters `/*` and runs until the next occurrence of `*/`. Nested comments are forbidden.

   Your task is to construct a regular expression for traditional C comments, one suitable for use in a lexer generator.

   **Before** you start, there is some notation you may find useful:

   - To indicate the union of every character in the alphabet, we can write $\Sigma$. For example, if the alphabet is $\{a, b, c, d, e\}$, then

     $$\Sigma = (a|b|c|d|e),$$

     when written in a regular expression.
   - We use the symbol $\neg$ to indicate not. For example, if the alphabet is $\{a, b, c, d, e\}$, then

     $$\neg\{a\} = (b|c|d|e), \quad \text{and} \quad \neg\{b, d, e\} = (a|c).$$

(a) When they see this problem for the first time, people often immediately suggest

$$/ * \ (\Sigma \,|\, \backslash n\,)^* \ * /$$

Explain why this regular expression would not make a lexer skip comments correctly. (Big hint: greedy).

It's greedy. Let's say you have multiple comments in your code like so:

```
/* Here is some code: */
code
.
.
.
code
/* No more code. */
```

The regular expression given will consider **everything** to be part of one big comment. This is because it will read all of the $*/$ and $/*$ between the first and last ones as simply characters in the comment, matching them to the expression $(\Sigma \,|\, \backslash n\,)^*$. Since it is greedy, it will grab the biggest comment possible, and end up grabbing everything in the file.

(b) Once the problem with the previous expression is noted, most folks decide that comments should contain only characters that are not stars, plus stars that are not immediately followed by a slash. This leads to the following regular expression:

$$/ * \ (\neg\{*\} \,|\, * \neg\{/\}\,)^* \ * /$$

Find a legal 5-character C comment that this regular expression fails to match, and a 7-character ill-formed (non-valid-comment) string that the regular expression erroneously matches.

A 5-character C comment that is not matched:

$$/ * * * /$$

This is not matched because the inside of the comment is '*', which is not either

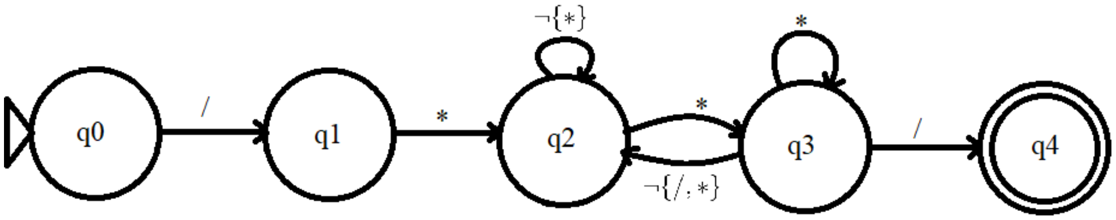
$$\neg\{*\} \quad \text{or} \quad * \neg\{/\}$$

A 7-character ill-formed string that is erroneously matched:

$$/ * * * / * /$$

This is matched because the inside of the "comment" is $**/$, which can be broken down as $*/$ (matching to $*\neg\{/\}$), followed by $/$ (matching to $\neg\{*\}$).

(c) Draw an NFA that accepts all and only valid traditional C comments.



Note: in this machine, states $q_0$ and $q_1$ represent code that isn't in a C comment.

(d) Provide a correct regular expression that describes all and only valid traditional C comments, by converting your NFA into a regular expression. Show all of your work.

NOTE: be sure it's completely clear where you are using the character $*$ and when you are using the regular expression operator $^*$.

Step 1: Note that we only have a single start and accepting states, so it already is a generalized NFA.

Step 2: start eliminating states.

- If we eliminate $q_1$, the transition from $q_0 \to q_2$ becomes $/*$.
- If we eliminate $q_3$, the transition from $q2$ to itself becomes:

  $$(\,\neg\{*\}\,|\,[*]^{+}\neg\{/,*\}\,)$$

  while the transition from $q_2 \to q_4$ becomes

  $$[*]^{+}/$$

  Note that $*[*]^* = [*]^+$.

- Finally, eliminating $q_2$ gives us the regular expression for all and only valid C comments:

  $$/\,*\,\,(\,\neg\{*\}\,|\,[*]^{+}\neg\{/,*\}\,)^*\,\,[*]^{+}/$$