Math 590 Homework 2 Fall 2018

Due: Thursday, September 20

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.

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

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

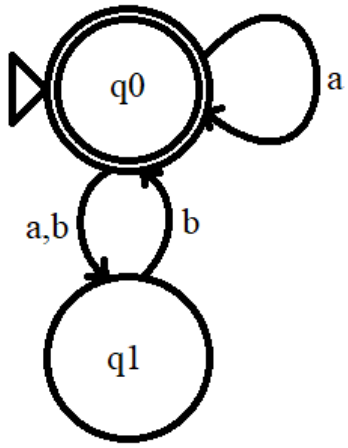
   (d) The language $1^*(001^+)^*$, 3 states.

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

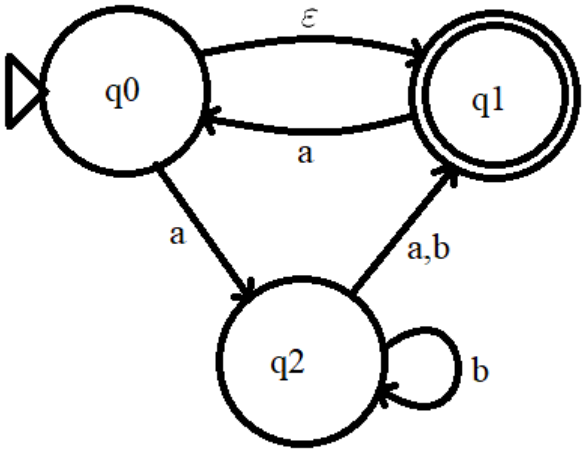
2. **Subset Construction**

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

(a) Prove that the set of regular languages are closed under the reverse operation, i.e., prove that if language A is regular, then $\mathit{reverse}(A)$ 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?)

(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 Σ_3 contains eight size-3 columns of 0s and 1s. A string of symbols in Σ_3 thus builds threes rows of 0s and 1s. Consider each row to be a binary number and define the language

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

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 $\mathit{reverse}(B)$ is regular, and by the previous problem conclude that $\mathit{reverse}(\mathit{reverse}(B)) = B$ is also regular.)

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

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

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

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

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

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

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 Σ. 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).

(continued)

(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:

$$/* \ (\neg\{*\} \mid * \neg\{/\})^* \ */$$

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.

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

(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 *.