Math 590 Homework 3 Solutions Fall 2019

Due: Thursday, September 19

1. **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) $\{w \mid w \text{ does not contain the substring 11}\}$

   $$(0|10)^*(\varepsilon|1) \quad \text{or} \quad 0^*(10^+)^*(\varepsilon|1)$$

   (e) $\{w \mid w \text{ has a length of at least 3 and its third symbol is a 0}\}$

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

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

   $$(\,1(0|1)\,)^*\,(\varepsilon|1) \quad \text{or} \quad (\,(\,1(0|1)\,)^*\,1\,|\,(\,1(0|1)\,)^*\,)$$

2. **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).

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

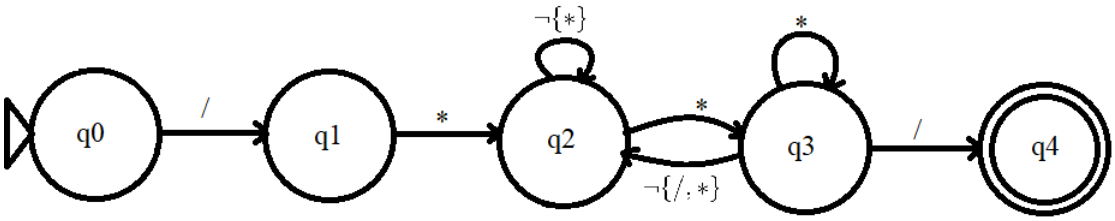
$$\neg\{*\} \quad \text{or} \quad * \neg\{/\}$$

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.



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

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

  while the transition from $q_2 \rightarrow q_4$ becomes

  $$\{*\}^+/$$

  Note that $*\{*\}^* = \{*\}^+$.

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

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

3. **Powers of 2**

   Describe an algorithm for a Turing Machine that decides the language consisting of all strings of 0s whose length is a power of 2:

   $$\{0^{2^n} \mid n \geq 0\}.$$

   You may assume that the input alphabet in this case is $\Sigma = \{0\}$.

   Solution #1:

   (a) If the input is empty, reject.

   (b) If there is only a single 0, accept.

   (c) Go through the input and cross off every other 0.

   (d) If there were an odd number of 0s, reject. Else, return to step (b).

   Solution #2:

   (a) If the input is empty, reject.

   (b) If there is only a single 0, accept.

   (c) Mark the first unmarked 0 and cross off the last 0.

   - If there was no last 0 to cross off, reject.
   - If there are no unmarked 0s, unmark all marked 0s and return to step (b).

4. **Ones and Zeros**

   Describe an algorithm for a Turing Machine that can decide the language consisting of all strings of n 1s followed by n 0s:

   $$\{1^n 0^n \mid n \in \mathbb{N}\}.$$

   You may assume that the input alphabet is $\Sigma = \{0, 1\}$.

   Solution #1:

   (a) If input is empty, reject. Otherwise, find the first symbol.

   (b) Read the first symbol.

   - If it is a $\sqcup$, accept.
   - If it is a 0, reject.
   - If it is a 1, replace with $\sqcup$.

   (c) Find and read the last symbol.

   - If it is a $\sqcup$, reject.
   - If it is a 0, replace with $\sqcup$.
   - If it is a 1, reject.

   (d) Find the first symbol, goto step (b).

Solution #2:

(a) If input is empty, reject. Otherwise, find the first symbol.

(b) Read the first non-a symbol.

- If it is a 0, reject.
- If it is a 1, replace with a.
- If it is a b, find the end of the string and accept if b, reject otherwise.

(c) Find and read the next non-1 symbol.

- If it is a 0, replace with b, goto step (e).
- If it is a $\sqcup$, reject.
- If it is a b, continue to step (d).

(d) Find and read the next non-b symbol.

- If it is a 0, replace with b.
- If it is a $\sqcup$, reject.
- If it is a 1, reject.

(e) Find the first symbol, goto step (b).

Note: for this solution, we could have used marked 0s and 1s instead of a and b. Effectively, the a was a marked 1, and the b was a marked 0.

Solution #3:

(a) Scan to input to ensure that it is 1s followed by 0s:

- If first symbol is $\sqcup$ or 0, reject. If 1, continue.
- Find first non-1. If $\sqcup$, reject. If 0, continue.
- Find first non-0. If 1, reject. If $\sqcup$, find start of string and goto step (b).

(b) Find first unmarked 1 and mark it.

- If no unmarked 1s, find last symbol.
- Accept if marked 0, reject if unmarked 0.

(c) Find first unmarked 0 and mark it.

- If no unmarked 0s, reject.

(d) Find start of string and goto step (b).

Note: this third solution was meant to illustrate that we are capable of scanning an input to ensure that it has the proper form.