Regular Expressions – Intermediate (basic level)
Exercises B – Gabor

Determine a regular expression which will match only on the indicated strings, or else will find the indicated matches. The form for the submission will be a command line script that takes a single integer as input from 40 to 49, inclusive, and outputs the corresponding regular expression pattern, just as with the prior HW. The pattern is to be delimited by forward slashes and any options should immediately follow the final slash.

For problems 40-42, an Othello board is any string of length 64 made up of only the characters in `"xX.Oo"`. An Othello edge is any string of length 8 made up of only the charaters in `"xX.Oo"`. A hole means a period. Actual Othello board are more restrictive, but this definition will suffice for the lab's purposes. `"x"` is synonymous with `"X"` and `"o"` is synonymous with `"O"` for the purposes of this problem.

40. Write a regular expression that will match on an Othello board.

41. Given a string of length 8, determine whether it could represent an Othello edge with exactly one hole.

42. Given an Othello edge, determine whether there is a hole at either end or, starting from either end, a contiguous sequence of (at least one) X which abuts a contiguous sequence (possibly empty) of O, which abuts a hole.

43. Match on all strings of odd length.

44. Match on all odd length binary strings starting with `0`, and on even length binary strings starting with `1`.

45. Match all words having two adjacent vowels that differ.

46. Match on all binary strings which DON'T contain the substring `110`.

47. Match on all non-empty strings over the alphabet {`a`, `b`, `c`} that contain at most one `a`.

48. Match on all non-empty strings over the alphabet {`a`, `b`, `c`} that contain an even number of `a`'s.

49. Match on all positive, base 3, integer strings that are even.

Regular Expressions
Intermediate – basic level

There is a single construct, mightily overloaded, which brings incredible power to regular expressions. The construct starts out straightforwardly, as a matched pair of parens, and works like you would expect for grouping. This gives us two immediate features at this point. The first is that we can have ORs within parentheses that apply locally, within the parens, while the second feature is that quantifiers can be applied to parens.

Thus, `/^0$|^10[10]$/` is equivalent to `/^(0|10[10])$/`

While in the above example, the lengths are the same, parentheses often bring an organizational clarity to regular expressions. The example below, which matches words of at least two vowels, is more compelling:

`/\w*[aeiou]\w*[aeiou]\w*/` is better written as `/(\w*[aeiou]){2}\w*/`

The quantifier on a parenthesized subexpression is the same as if that subexpression appeared as many times as the quantifier indicated. In the case of a question mark (one or zero), if the subexpression matches, that is preferred (greediness), but it is also OK if there is no match. Just as before, if the quantity that applies is more than one, there is no implication that there is any repetition. For example `/[aeiou]{2}/` does not say "repeated vowel". Rather, it says "adjacent vowels" because it is like writing `/[aeiou][aeiou]/`.

It is incorrect to think of parentheses as merely being a convenience, as they were in the prior two examples. In particular, parens contain subexpressions to which quantifiers may be applied. For example, `/^(abc?)+$/` says that it is looking for strings consisting of `ab` repeated, each of which may have up to one `c` following it. This regular expression can't be achieved without parens.

Here is another example that is impossible without parentheses: Match all strings consisting only of the characters in "ab" where the number of b's is a multiple of 3.