Regular Expressions – part 3 of 5 (Intermediate to Advanced Intermediate)

AI 2023-24

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 50 to 59, inclusive, and outputs the corresponding regular expression pattern, just as with prior HW. The pattern is to be delimited by forward slashes and any options should immediately follow the final slash.

For problems 50-52 no look ahead or look behind is allowed. For the remaining problems, you may use the full power of Python's regular expressions. For all the problems mentioning words (rather than strings), substrings and other comparisons are meant to be case insensitive unless otherwise specified. With strings, the default is case sensitive.

50. Match all words where some letter appears twice in the same word.

51. Match all words where some letter appears 4 times in the same word.

52. Match all non-empty binary strings with the same number of `01` substrings as `10` substrings.

53. Match all 6 letter words containing the substring `cat`.

54. Match all 5 to 9 letter words containing both the substrings `bri` and `ing`.

55. Match all 6 letter words not containing the substring `cat`.

56. Match all words with no repeated characters.

57. Match all binary strings not containing the forbidden substring `10011`.

58. Match all words having two different adjacent vowels.

59. Match all binary strings containing neither `101` nor `111` as substrings.

Regular Expressions – All about parens

Parens remember the most recent substring that they munched. This is known as capturing, and these captures can be referred to within the regular expression using back references.

Parens are numbered, left to right, starting from 1, based on the opening paren. This implies that if a paren is nested within another, the nested paren has the higher number. Parens are numbered unless they are immediately followed by `?`, which is the escape symbol in a parenthesized expression.

A back reference is a construct of the form `\1`, `\2`, `\3`, … When a back reference appears, it must match exactly whatever the indicated paren captured. If the indicated paren has not yet captured anything, then the back reference is undefined (and the entire regular expression may break).

`(?:...)` is like a normal paren, except that it is unnumbered (ie. it doesn't remember what it captured).

`(?=...)` is a look ahead (an example of a look around). It is an assertion and does not munch, insisting that there must be a match according to the pattern that the ellipsis represents, at that exact point. If this is not the case, then the regular expression outside the lookahead may backtrack and try to find another branch. Parens inside the look ahead may capture and remember, but the lookaround parens themselves do not. An example where lookaheads are helpful is: match on all decimal strings that have at least two 5s and at least three 4s: `/^(?=.*5.*5)(\d*4){3}\d*$/`, which tells the regex engine to look ahead and ensure that there are at least two 5s, and then to munch digits all the way to the end ensuring that there are at least three 4s.

`(?!...)` is a negative look ahead. It is an assertion and does not munch, insisting that there must not be a match according to the pattern that the ellipsis represents, at that exact point. If there is a match, then the negative lookahead fails, and the regular expression outside the negative lookahead may backtrack and try to find another branch. Parens inside a negative lookahead can be back referenced within the negative lookahead, but it makes no sense outside of that context.

An example where negative lookahead is handy is to find a regex for matching on words that start and end with different vowels. It might look like: `/\b([aeiou])\w*(?!\1)[aeiou]\b/i`, which says that the starting letter is a vowel (which should be remembered as `\1`), and then all remaining letters should be munched through the penultimate one, and the final letter should not be the initial one, and then the final letter, which must be a vowel, should be munched. Note that the `\b` is necessary because otherwise the regex engine will find any two vowels that work such as owe in snowed.

If you want to do an *and* with lookaheads, it should be done as (?=condition1)(?=condition2) while an *or* would be done as (?=condition1|condition2). If you want to ensure the absence of two conditions, then use (?!condition1|condition2), which says stop if either condition1 or condition2 holds.

`(?<=...)` and `(?<!...)` are look behind and negative look behind constructs, but they are severely restricted in that they must know ahead of time exactly how many characters they refer to (ie. be fixed width). That means no variable quantifiers, though back references are tolerated. Not all languages (eg. legacy javascript) support this construct.