HW 3 Solution
Regular Expressions and Finite Automata

CS 440: Programming Languages and Translators

Was due Sat Feb 16

1. (English descriptions for the languages of regular expressions)
   1a. Dates of the form year-day-month where the years are 1900 – 2099, with leading zeros optional on the days and months. (There’s no check for which months have 28, 30, or 31 days, so it permits 2019-02-29, for example.)
   1b. #17: [1-9a-h] [0-9a-h]* (e [+-]? [1-9]\d*) — an unsigned base 17 number (lower-case letters a-f only), with no leading zeros, followed by an optional exponent (letter e, an optional sign, and an unsigned decimal number (with no leading zeros).

2. (Write some regular expressions)
   2a. (Strings with an even # a’s or number of b’s divisible by 2): (b* a b* a)* b* | (a* b a* b a* b) * a*
      For an even # a’s, look at sequences with exactly two a’s and repeat them. We have to be sure that the string can begin and end with an arbitrary number of b’s. Similarly, for the b’s, repeat sequences with exactly three b’s, with the string beginning or ending with an arbitrary number of a’s.
      Note: Since concatenation is associative, answers like b* ((a b* a) b*) * | ....are possible too.

2b. (Strings that don’t contain the 3-character substring abc): ([^a] | a[^b] | ab[^c]) *[^abc]*
      Here, we concentrate on the substrings a, ab, and abc (an improper substring). Our overall string is a sequence of things that don’t include any of those three: to avoid a a, we look for not a; to avoid ab, we look for not b; to avoid abc, we look for ab not c. Then, once we see the last c in our string, we can have any number of a’s and b’s.

2b. The TA pointed out that my solution had a bug: a [^b] followed by [^a] and [^a] again can match aabc, which contains abc. The easiest solution to understand is probably the DFA one: You build a DFA that accepts only abc and then flip all the accepting/nonaccepting statuses so that abc is the only string you reject. Although there is indeed a regular expression for this DFA, it’s messy, so for grading purposes, as long as you made a game attempt at solving the problem, you’ll get full credit.

![DFA that rejects abc](image-url)
3. (Write regular expressions for numerals)
   3a. Natural numbers with no leading zeros (except for just 0): 0 | [1-9] \d*
   3b. Integer with optional sign:
       Without leading zeros: [+–] natnum exponent? where exponent is e [+–] natnum
       With leading zeros: [±] \d+ exponent? recall (exp + is exp exp*)
   3c. Floating point numbers:
       Without leading zeros: (natnum . \d* | . \d+) exponent?
       With leading zeros: (\d+. \d* | . \d+) exponent?
   3d. Hex numbers allowing leading zeros: 0x [0-9a-fA-F] +

4. (Reg expr to NFA)
   Here is a pretty mechanical translation of the expression using the algorithm in the text. (You weren't required to have exactly this. See the next page for something more understandable.)

((b | c) a (a | c))* before removing any redundant ε arrows
5. (Convert NFA to DFA) Here's the original NFA.

Original NFA
First we take the $\epsilon$-closure of the NFA to get one without $\epsilon$-transitions (the discussion after Figure 2.6 is useful here).

![\epsilon$-closure of NFA](image)

Then get rid of any stuck paths by adding an error state and transitions to it.

![After Adding AN Error State](image)
Then use the DFA-state $= \text{set of NFA-states}$ technique to complete the conversion (the discussion before Figure 2.3 is useful here).