Homework 3

6CCS3CFL - Compilers & Formal Languages

Finley Warman

November 1, 2020

Contents

Question 1

Q: The regular expression matchers in Java, Python and Ruby can be very slow with some (basic) regular expressions. What is the main reason for this inefficient computation?

A: These languages construct an equivalent NFA to the input regular expression ‘under the hood’. Translating these expressions into NFAs can cause the size of the resulting automaton to grow (typically O(n) space). Testing to see if the automaton accepts the input string is done often with a depth-first search, which can be slow for these large inputs as backtracking is required if visiting a ‘wrong’ path to an accepting state.

Question 2

Q: What is a regular language? Are there alternative ways to define this notion? If yes, give an explanation why they define the same notion

A: A regular language is a language that can be expressed by a regular expression. This is equivalent to a language that can be described with a finite automaton (NFA or DFA). These notions are the same, since all NFAs can be converted to equivalent DFAs, and we know from Thompson that a regular expression can be converted to a NFA.

Question 3

Q: Why is every finite set of strings a regular language?

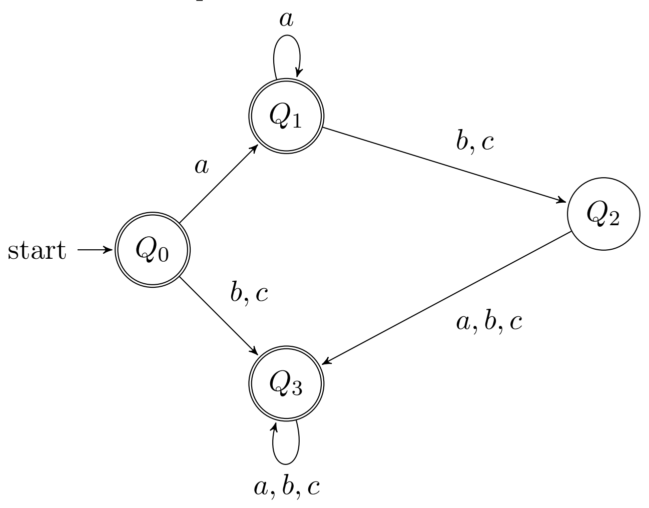
A: A high-level explanation: for any finite set of strings, we can simply construct an NFA with one ’branch’ leading to a linear DFA for each string in the language.
We can also say that since all single strings are regular, and any union of two regular languages is regular, then this finite set of strings can be constructed from the union of the languages containing each string, and is therefore regular.

Question 4

Q: Assume you have an alphabet consisting of the letters a, b and c only.
(1) Find a regular expression that recognises the two strings ab and ac.
(2) Find a regular expression that matches all strings except these two strings.

A: 1) `(a.(b+c))`
A: 2) `a* + (a.a*.(b+c).(a+b+c)*) + ((b+c).(a+b+c)*)`

NFA for this expression:

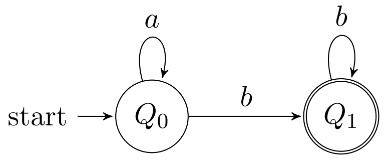


Question 5

Q: Given the alphabet {a,b}. Draw the automaton that has two states, say Q_0 and Q_1. The starting state is Q_0 and the final state is Q_1.
The transition function is given by:

```
(Q0,a)→Q0
(Q0,b)→Q1
(Q1,b)→Q1
```

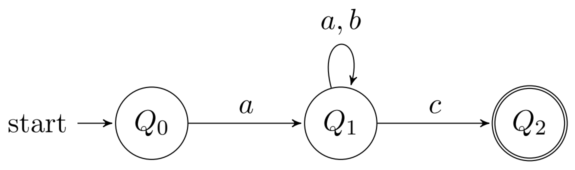
A: Finite State Machine:



What does this language accept?
Any number of ‘a’ followed by 1 or more ‘b’. (`a*b+`)

Question 6

Q: Give a non-deterministic finite automaton that can recognise the language `L(a·(a+b)*·c)`



Question 7

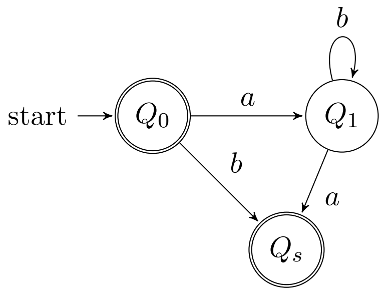
Given a DFA $A(\Sigma, Q, Q_0, F, \delta)$, the language accepted by this DFA, L, is described as $L = \{s \mid \hat{\delta}(Q_0, s) \in F\}$ - that is, all strings s which the automaton accepts (there is a set of transitions for each consecutive character in the string to some accepting state).

For an NFA $A(\Sigma, Q_s, Q_{0s}, F, \rho)$, the language L is $L = \{s \mid \hat{\rho}(Q_{0s}, s) \in F\}$. (All strings accepted by the NFA)

Question 8

Q: Find an automaton recognising complement language, over alphabet {a, b}

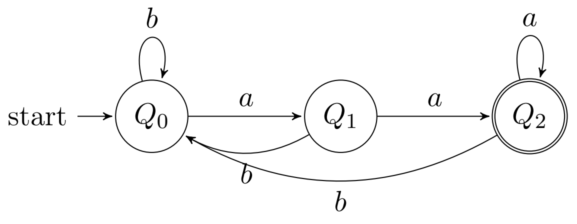
A: To convert an automaton to its complement, first 'complete' it (e.g. by adding a 'sink state' for all missing transitions), then swap all accepting and non-accepting states:



Question 9

Q: Convert NFA over {a,b} to DFA:

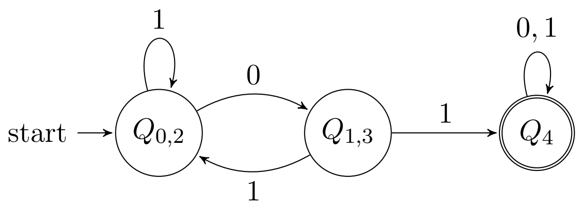
A:



Question 10

Q: Find corresponding minimal automaton for DFA over {0,1}.

A:



Question 11

Q: Give a regular expression recognising the same language as DFA over {a,b}.

A: `(b+ab+aa(a*)b)*aa(a)*`

Question 12

Q: How many (maximum) states does a DFA need to match an equivalent DFA with n states?
A: 2^n

Question 13

Q: Prove for all regular expressions r we have $nullable(r)$ iff $[] \in L(r)$
nullable definition:

```
nullable(r): match r => Boolean
  case 0        == false
  case 1        == true
  case c        == false
  case r1+r2    == nullable(r1) || nullable(r2)
  case r1.r2    == nullable(r1) && nullable(r2)
  case r*       == true
```

For expressions of the forms **0** and c, nullable(r) does not hold, and $[] \notin L(r)$.

For expressions of the form **1**, $[] \in L(r)$ holds by definition.

In the case of r^*, $[] \in L(r)$ since r^* matches 0-or-more of r, which includes the empty string.

The final two cases are defined recursively in terms of nullable(r), and possible cases are covered as above.

For an expression r of the form $r_1 + r_2$, then either r_1 or r_2 must be nullable for $nullable(r)$ to hold. For an expression r of the form $r_1.r_2$, then both r_1 and r_2 must be nullable for $nullable(r)$ to hold.
As from the above defintions, we know that if nullable(r) holds for these two cases, then $[] \in L(r)$, otherwise, $[] \notin L(r)$.