KØBENHAVNS
UNIVERSITET

Diskret Matematik og Formelle Sprog: Problem Set 5

Due: Wednesday March 30 at 09:59 CET.
Submission: Please submit your solutions via *Absalon* as PDF file. State your name and e-mail address close to the top of the first page. Solutions should be written in LaTeX or some other math-aware typesetting system with reasonable margins on all sides (at least 2.5 cm). Please try to be precise and to the point in your solutions and refrain from vague statements. Make sure to explain your reasoning. *Write so that a fellow student of yours can read, understand, and verify your solutions.* In addition to what is stated below, the general rules for problem sets stated on *Absalon* always apply.
Collaboration: Discussions of ideas in groups of two to three people are allowed—and indeed, encouraged but you should always write up your solutions completely on your own, from start to finish, and you should understand all aspects of them fully. It is not allowed to compose draft solutions together and then continue editing individually, or to share any text, formulas, or pseudocode. Also, no such material may be downloaded from the internet and/or used verbatim. Submitted solutions will be checked for plagiarism.
Grading: A score of 120 points is guaranteed to be enough to pass this problem set.
Questions: Please do not hesitate to ask the instructor or TAs if any problem statement is unclear, but please make sure to send private messages — sometimes specific enough questions could give away the solution to your fellow students, and we want all of you to benefit from working on, and learning from, the problems. Good luck!

1 (60 p) Consider the regular expression

$$a^*(b|c)\big(d^*|ef\big)^*$$

and determine which of the words below belong to the language generated by this regular expression. Motivate your answers briefly but clearly by explaining for each string how it can be generated or by arguing why this is impossible.

1. *defdefdef*
2. *aaaac*
3. *cdef*
4. *aaabcef*
5. *aabddeeff*
6. *aabdddefefd*

Solution: We give 10 p per fully correct and satisfactorily motivated answer. It stands to reason that any such answer will have to involve a discussion of how how to interpret the regular expression (in order to argue why accepted strings are accepted), so let us start by explaining this.

The regular expression $a^*(b|c)\big(d^*|ef\big)^*$ accepts words on the following form:

- First comes a string of zero or more characters a.
- Then follows exactly one b or one c.
- Finally, zero or more times the following pattern repeats:
  - zero or more characters d, or
  - the string ef.

With this in mind, we get the following answers:

1. $defdefdef$ ✗ (There is no b or c.)
2. $aaaac = a^4c\big(d^*|ef\big)^0$ ✓
3. $cdef = a^0cd^1(ef)^1$ ✓
4. $aaabcef$ ✗ (Both b and c cannot appear.)
5. $aabddeeff$ ✗ (Cannot have two e or two f in a row.)
6. $aabdddefefd = a^2bd^3(ef)^2d^1$ ✓

2 (90 p) Consider the following context-free grammars, where a, b, c are terminals, B, S are non-terminals, and S is the starting symbol.

Grammar 1:

$$S \to aS \tag{1a}$$
$$S \to B \tag{1b}$$
$$B \to bcB \tag{1c}$$
$$B \to \tag{1d}$$

Grammar 2:

$$S \to aS \tag{2a}$$
$$S \to BS \tag{2b}$$
$$S \to B \tag{2c}$$
$$B \to bcB \tag{2d}$$
$$B \to \tag{2e}$$

Grammar 3:

$$S \to aS \tag{3a}$$
$$S \to BS \tag{3b}$$
$$S \to B \tag{3c}$$
$$B \to bBc \tag{3d}$$
$$B \to \tag{3e}$$

Which of these grammars generate regular languages? For each grammar, write a regular expression that generates the same language (and explain why), or argue why the language generated by the grammar is not regular. In your regular expressions, please use *only* the concatenation, alternative (|) and star (*) operators, and not the syntactic sugar extra operators that we just mentioned briefly in class but never utilized.

Solution: We give 30 p per correct and satisfactorily motivated answer.

Grammar 1 generates strings that contain zero or more occurrences of a (obtained from S) followed by zero or more occurrences of bc (generated from B once the production $S \to B$ has been used). This languages is captured by the regular expression $a^*(bc)^*$.

Grammar 2 is very similar, except that the production $S \to BS$ means that after having generated any string in the language $L(G_1)$, i.e., any string matching Grammar 1, we can start over with the symbol S and generate another copy if we like. In other words, we get a language $L(G_2) = \{w_1 w_2 \cdots w_n \mid w_i \in L(G_1),\, n \in \mathbb{N}\}$ consisting of concatenations of zero or more copies of strings matching Grammar 1. This languages is captured by the regular expression $\bigl(a^*(bc)^*\bigr)^*$.

The language generated by Grammar 3 is not regular. To see this, note that strings of the form $b^n c^n$ are in the language while strings of the form $b^n c^{n+1}$ are not. In order to distinguish between the two, the regular expression would need to do counting, but as mentioned in Mogesen's notes and during the lectures this is something that regular expressions cannot do. The easiest way to see this is to use the fact that any language generated by a regular expression can also be recognized by a deterministic finite automaton (DFA), but DFAs have a finite number of states and so cannot possibly do unlimited counting. (You do not have to prove a formal version of the claim that "regular expressions cannot count" in order to get full credits for this problem referring to what is said in the notes or what we covered in class is enough.)

3 (80 p) In this problem, we want to write regular expressions and context-free grammars generating specified languages.

3a (40 p) Write a regular expression for the language consisting of all finite (possibly empty) bit strings (i.e., over the alphabet $\{0,1\}$) that do not contain any consecutive 0s. Examples of such strings are ε (the empty string), 010, 10110111, and 111, whereas 100 and 010010 do not qualify for membership. For partial credit (if your regular expression is wrong or missing), argue why this language is clearly regular.

Solution: Let us first note that to argue that this language is regular, it is sufficient to build a deterministic finite automaton that recognizes it. Such an automaton could consist of two states S_0 and S_1 with S_0 being the starting state and both states being accepting. The intuition is that in state S_0 we can accept reading 0 but in state S_1 we insist on a 1. Formally, the transitions are as follows:

- Both S_0 and S_1 have a transition to S_0 on 1.
- S_0 has a transition to S_1 on 0.
- S_1 does *not* have any transition on 0.

For any string consisting two 0s in a row the walk in the DFA will get stuck. All other strings will be accepted, since all states are accepting. See Figure 1 for an illustration.

If we wanted to, we could also have a solution with a third state S_3, which is non-accepting, such that S_1 has a transition to S_2 on 0 and all transitions from S_2 are to S_2.

To construct a regular expression for this language, let us do a case analysis and write regular expressions for each case. Once the case analysis is finished, we can combine all the cases with the union operator $|$. We have the following cases:

- The string could consist of zero or more 1s only and no 0s. This is captured by the regular expression 1^*.

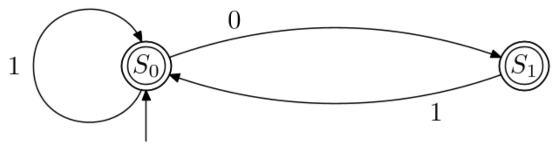


Figure 1: DFA for bit string without consecutive zeros in Problem 3a.

- Otherwise the string is non-empty and contains at least one 0. We will split the string up into three parts before the first 0, up to the last 0, and after the last 0 (where we note that the last two parts could be empty). If we write down regular expressions for these three parts, they can then be combined with concatenation. We get the following:

  1. The part of the string up to and including the first 0 is described by the regular expression 1^*0.
  2. If the string contains more 0s, every 0 is preceded by one or more 1s, which matches $(11^*0)^*$.
  3. Finally, after the last 0 we have zero or more 1s, i.e., 1^*.

  Concatenating these expressions yields $1^*0(11^*0)^*1^*$.

Taking the union of the two regular expressions obtained at the top level of our case analysis yields the final answer $1^* \mid \bigl(1^*0(11^*0)^*1^*\bigr)$. Other solutions than this are also possible, of course.

3b (40 p) Give a context-free grammar for the language $\bigl\{(a|b)^n(c|d)^n \bigm| n \in \mathbb{N}\bigr\}$. Examples of strings in this language are ε (the empty string), *aaacdc*, and *abbadddc*, whereas *abc*, *ababcdcdc*, and *abccddba* do not make the cut.

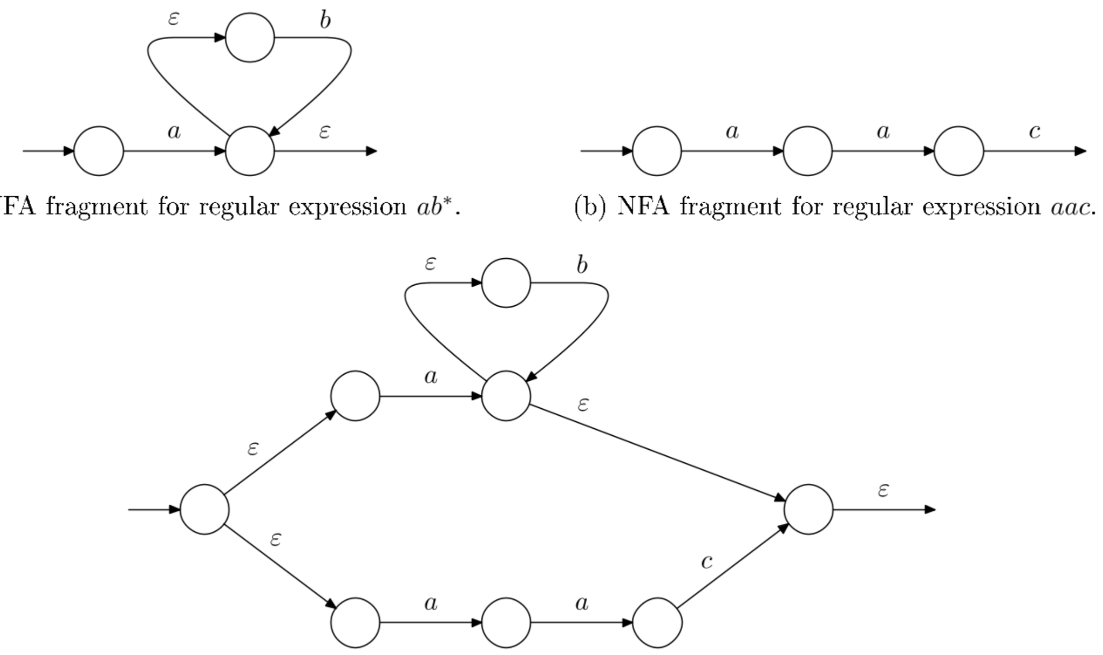
Solution: Strings in this language should start with zero or more occurrences of a or b, with every occurrence being matched by an occurrence of either c or d at the end. A language as described above can be generated by, e.g., the following grammar:

$$
\begin{aligned}
&S \to USV \\
&S \to \varepsilon \\
&U \to a \\
&U \to b \\
&V \to c \\
&V \to d
\end{aligned}
$$

Each non-terminal U will generate a or b and each V will generate c or d. The first production $S \to USV$ ensures that the same numbers of Us and Vs are generated, with the Us preceding the Vs.

4 (120+ p) In this problem we want to construct a deterministic finite automaton (DFA) that recognizes the language generated by the regular expression $((ab^*)|(aac))^*$.

4a (60 p) Translate this regular expression to a nondeterministic finite automaton (NFA) using the method from Mogensen's notes that we learned in class. Make sure to describe clearly which part of the regular expression corresponds to which part of the NFA, and which states are accepting. Please do not take any shortcuts without explaining what you are doing.



(a) NFA fragment for regular expression ab^*.

(b) NFA fragment for regular expression aac.

(c) NFA fragment for regular expression $ab^*|aac$.

Figure 2: NFA fragments in translation of regular expression in Problem 4.

Solution: We follow the procedure outlined in Section 1.3 in Mogesen's notes. Three NFA fragments with corresponding regular (sub)expressions are shown in Figure 2. The full translation of the regular expression to a nondeterministic finite automaton is as in Figure 3, where we have added numbers to the states to be able to refer to them in the next subproblem.

As a side note, we remark that the expression $((ab^*)|(aac))^*$ is overly paranthesized, just to make the intended meaning clear. Since we agreed in class that the star operator binds hardest, followed by concatenation, it would have been fully sufficient to just write $(ab^*|aac)^*$.

4b (60 p) Translate the nondeterministic finite automaton to a deterministic finite automaton using the method from Mogensen's notes that we learned in class. Make sure to explain in detail how you perform the subset construction, so that it is possible to follow your line of reasoning. Present clearly the resulting DFA, with all states and transitions, and specify which states are accepting.

Solution: We follow Algorithm 1.3 in Section 1.5.2 of Mogesen's notes, except that we use calligraphic letters $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ... to refer to the constructed states in the deterministic finite automaton (to distinguish them more clearly from the NFA states). Referring in what follows to the numbered states in the NFA in Figure 3, the starting state is

$$\varepsilon\text{-closure}(\{1\}) = \{1, 2, 3, 6, 10\} = \mathcal{A}$$

(where we recall that the ε-*closure* of a set of states S is any state that can be reached from some state in S via zero or more ε-transitions). The possible transitions from $\mathcal{A}$ on characters

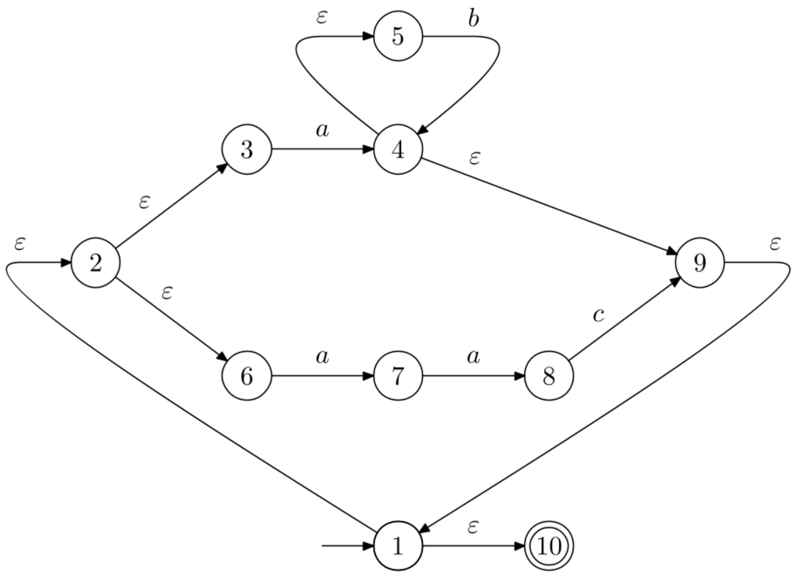


Figure 3: Full translation to NFA of regular expression in Problem 4.

a, b, and c are

$\text{move}(\mathcal{A}, a) = \varepsilon\text{-closure}(\{4, 7\}) = \{1, 2, 3, 4, 5, 6, 7, 9, 10\} = \mathcal{B}$	[**new state**]
$\text{move}(\mathcal{A}, b) = \emptyset$	[no transition possible]
$\text{move}(\mathcal{A}, c) = \emptyset$	[no transition possible]

We consider next the new state $\mathcal{B}$, for which we get the transitions

$\text{move}(\mathcal{B}, a) = \varepsilon\text{-closure}(\{4, 7, 8\}) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} = \mathcal{C}$	[**new state**]
$\text{move}(\mathcal{B}, b) = \varepsilon\text{-closure}(\{4\}) = \{1, 2, 3, 4, 5, 6, 9, 10\} = \mathcal{D}$	[**new state**]
$\text{move}(\mathcal{B}, c) = \emptyset$	[no transition possible]

Moving on to state $\mathcal{C}$ we obtain

$\text{move}(\mathcal{C}, a) = \varepsilon\text{-closure}(\{4, 7, 8\}) = \mathcal{C}$	
$\text{move}(\mathcal{C}, b) = \varepsilon\text{-closure}(\{4\}) = \mathcal{D}$	
$\text{move}(\mathcal{C}, c) = \varepsilon\text{-closure}(\{9\}) = \{1, 2, 3, 6, 9, 10\} = \mathcal{E}$	[**new state**]

and for state $\mathcal{D}$ we derive

$\text{move}(\mathcal{D}, a) = \varepsilon\text{-closure}(\{4, 7\}) = \mathcal{B}$	
$\text{move}(\mathcal{D}, b) = \varepsilon\text{-closure}(\{4\}) = \mathcal{D}$	
$\text{move}(\mathcal{D}, c) = \emptyset$	[no transition possible]

Finally, for state $\mathcal{E}$ we can observe that it is identical to $\mathcal{A}$ except for the added NFA state 9, the only transition from which is an ε-transition to 1. We therefore should get the same transitions as for state $\mathcal{A}$, and indeed we can compute that

$$\text{move}(\mathcal{E}, a) = \mathcal{B}$$
$$\text{move}(\mathcal{E}, b) = \emptyset$$
$$\text{move}(\mathcal{E}, c) = \emptyset$$

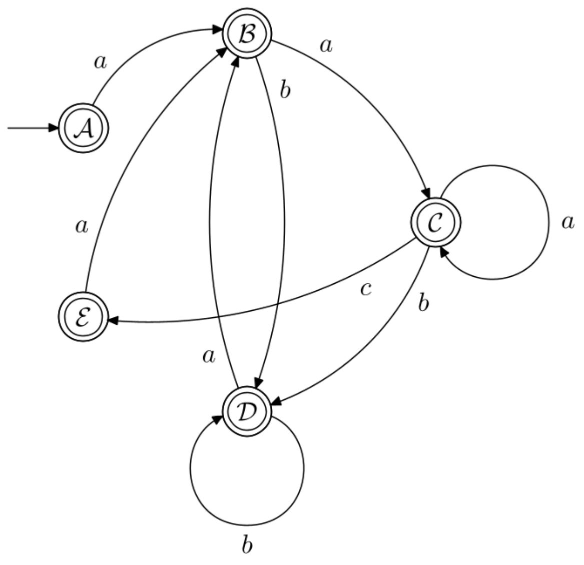


Figure 4: Conversion of NFA in Figure 3 to DFA.

Since we have now considered all possible transitions from all states in our constructed DFA, we are done. Interestingly, since all subsets of states contain the accepting NFA state 10, all states in our constructed DFA are accepting. Hence, the DFA will never reject a string by ending up in the "wrong state"—what will happen for illegal strings is that we will be in a state where there is no legal transition for the next character. See Figure 4 for an illustration of the constructed automaton.

4c *Bonus problem (worth 40 p extra; might be hard):* How small a DFA can you produce that accepts precisely the language generated by the regular expression above? Can you prove that the size of your DFA is optimal? Note that that you can solve this problem even if you did not solve the other subproblems above.

Solution: Looking at Figure 4, it is not hard to see that the states $\mathcal{A}$ and $\mathcal{E}$ have exactly the same outgoing transitions. We can therefore merge these two states to get a smaller DFA as in Figure 5. Let us now argue why any DFA for the regular expression $(ab^*|aac)^*$ must have at least 4 states, meaning that the construction in Figure 5 is optimal.

Observe first that we need 3 states keeping track of whether we have seen 0, 1, or 2 or more a's since the last read c (since this determines whether we are ready to accept another c or not). These are the states $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, in Figure 5.

Argued slightly differently (and in more detail), the initial strings (i) ε, (ii) a, and (iii) aa must lead to different states. First note that the states corresponding to these strings must all be accepting, since ε, a, and aa all match the regular expression. Now we can observe that (iii) is the only one of these strings after which we are willing to append a c to obtain aac, whereas c and ac are invalid strings. Hence, string (iii) must end up in a different state from that of strings (i) and (ii). Furthermore, after having seen (ii) we accept ac, resulting in the string aac, but we cannot append the same string to the empty string in (i) since ac is not a valid string. Therefore, strings (i) and (ii) must also end up in different states in the DFA.

To see why at least one more state is needed, consider the state of the DFA after having read ab. After this we are willing to accept another b to get abb. This rules out the initial state corresponding to (i), since if the DFA were in that state it would also have to accept the string b,

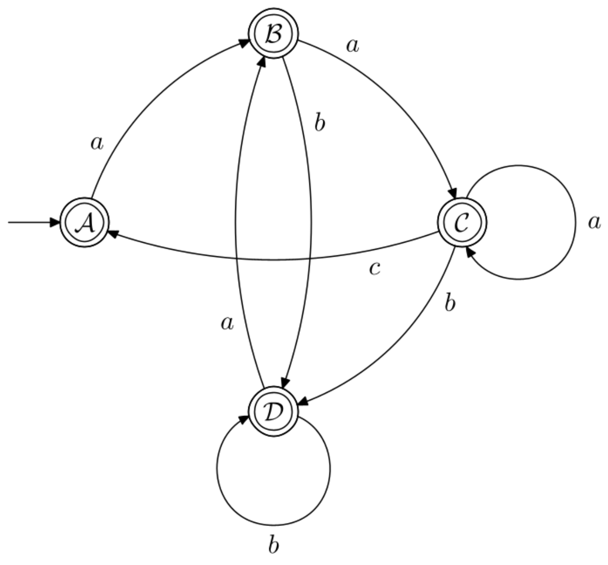


Figure 5: Smallest possible DFA equivalent to that in Figure 4.

which is not legitimate. Furthermore, after having read ab it is not acceptable to append ac, since $abac$ does not match the regular expression, and this rules out the state corresponding to (ii). Finally, since appending c to obtain abc is also not acceptable, we cannot be in the state corresponding to (iii). This shows that after having read ab the DFA must be in a different state than after having read strings (i), (ii), or (iii), and so an additional state like $\mathcal{D}$ in Figure 5 is also needed.

We remark that it is not necessary for this subproblem to go the formal route by first constructing an NFA from the regular expression and then converting this NFA to a DFA. Just by studying the regular expression and understanding what it means, it is possible to write down the DFA in Figure 5 directly, and then argue that the size has to be optimal.