

# Discrete Event Systems

## Solution to Exercise Sheet 3

### 1 Pumping Lemma [Exam]

**The Pumping Lemma in a Nutshell**

Given a language  $L$ , assume for contradiction that  $L$  is regular and has the pumping length  $p$ . Construct a suitable word  $w \in L$  with  $|w| \geq p$  (“there *exists*  $w \in L$ ”) and show that for *all* divisions of  $w$  into three parts,  $w = xyz$ , with  $|x| \geq 0$ ,  $|y| \geq 1$ , and  $|xy| \leq p$ , there *exists* a pumping exponent  $i \geq 0$  such that  $w' = xy^iz \notin L$ . If this is the case,  $L$  is not regular.

Language  $L$  can be shown to be non-regular using the pumping lemma. Assume for contradiction that  $L$  is regular and let  $p$  be the corresponding pumping length. Choose  $w$  to be the word  $0110^p1^p$ . Because  $w$  is an element of  $L$  and has length more than  $p$ , the pumping lemma guarantees that  $w$  can be split into three parts,  $w = xyz$ , where  $|xy| \leq p$  and for any  $i \geq 0$ , we have  $xy^iz \in L$ . In order to obtain the contradiction, we must prove that for every possible partition into three parts  $w = xyz$  where  $|xy| \leq p$ , the word  $w$  cannot be pumped. We therefore consider the various cases.

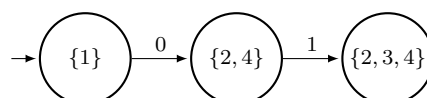
- If  $y$  starts anywhere within the first three symbols (i.e. 011) of  $w$ , deleting  $y$  (pumping with  $i = 0$ ) creates a word with an illegal prefix (e.g.  $10^p1^p$  for  $y = 01$ ).
- If  $y$  consists of only 0s from the second block, the word  $w' = xy^2z$  has more 0s than 1s in the last  $|w'| - 3$  symbols and hence  $c \neq d$ .

Note that  $y$  cannot contain 1s from the second block because of the requirement  $|xy| \leq p$ .

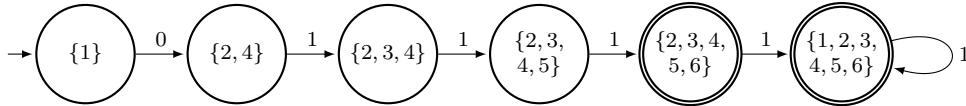
We have shown that for all possible divisions of  $w$  into three parts, the pumped word is not in  $L$ . Therefore,  $L$  cannot be regular and we have a contradiction.

### 2 Deterministic Finite Automata [Exam]

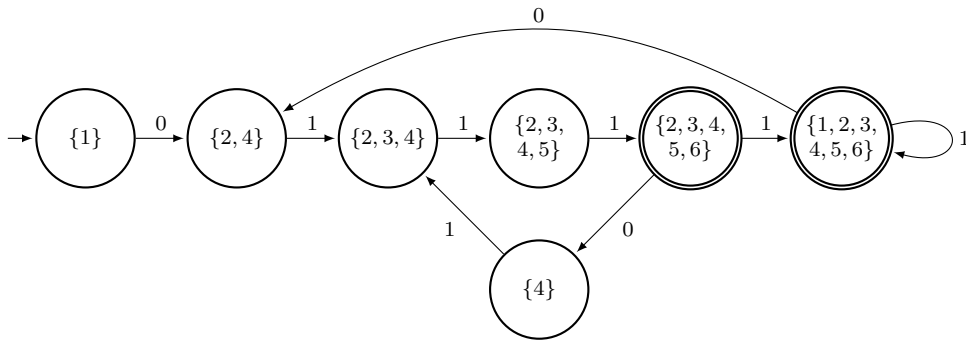
We could use the systematic transformation scheme presented in the lecture (slide 1/75). Considering the large number of states, however, this will easily lead to an explosion of states in the derandomized automaton. Hence, we build the deterministic finite automaton in a step-wise manner, only creating those states that are actually required: Initially, the automaton requires a 0. Subsequently, only a 1 is accepted. Including the various transitions, this 1 can lead to three different states, namely states 2, 3, and 4.



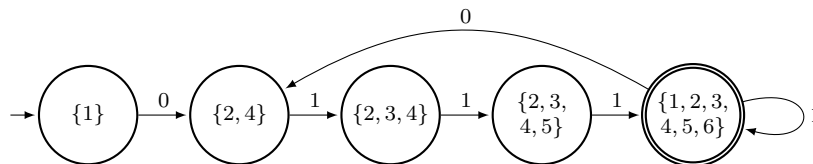
In any of the states 2, 3, and 4, only a 1 is accepted. Assume that the automaton is currently in state 2, this 1 can lead to states  $\{2, 3, 4\}$  when including all  $\varepsilon$ -transitions. When in state 3, the 1 leads to states  $\{2, 3, 4, 5\}$  and finally, when being in state 4, the reachable states given a 1 are  $\{2, 3, 4\}$ . Hence, a 1 leads from state  $\{2, 3, 4\}$  to state  $\{2, 3, 4, 5\}$ . Repeating the same process for state  $\{2, 3, 4, 5\}$ , we can see that, again, only a 1 is accepted, which leads to state  $\{2, 3, 4, 5, 6\}$ . Because the state 6 in the original NFA was an accepting state,  $\{2, 3, 4, 5, 6\}$  is also accepting in the DFA. From state  $\{2, 3, 4, 5, 6\}$ , an additional 1 will lead to another accepting state  $\{1, 2, 3, 4, 5, 6\}$ . And from this state, any subsequent 1 returns to state  $\{1, 2, 3, 4, 5, 6\}$  as well.



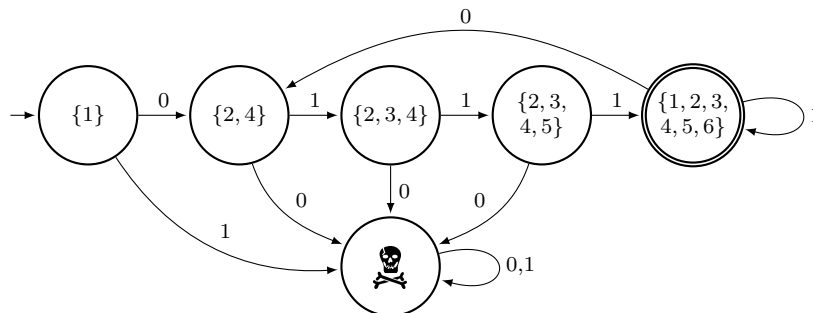
What happens if a 0 occurs in the input? This is feasible only when the deterministic state includes either state 1 or state 6. In state  $\{2, 3, 4, 5, 6\}$ , a 0 necessarily leads to state  $\{4\}$ , whereas in state  $\{1, 2, 3, 4, 5, 6\}$  a 0 leads to state  $\{2, 4\}$ . In both of these states, the only acceptable input symbol is a 1 and leads to the state  $\{2, 3, 4\}$ . Hence, the deterministic finite automaton looks like this:



It can easily be seen, that first the states  $\{4\}$ ,  $\{2, 4\}$  and then the states  $\{2, 3, 4, 5, 6\}$ ,  $\{1, 2, 3, 4, 5, 6\}$  can be merged and hence, the automaton can be reduced to the one shown in the next figure.

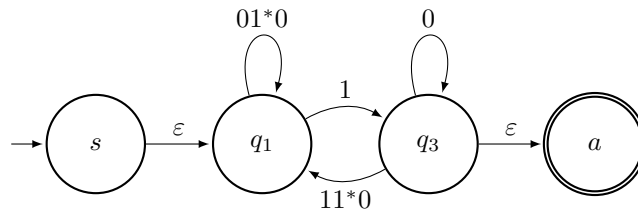


This is not a DFA yet, because the crash state is still missing. The final deterministic automaton looks like this:

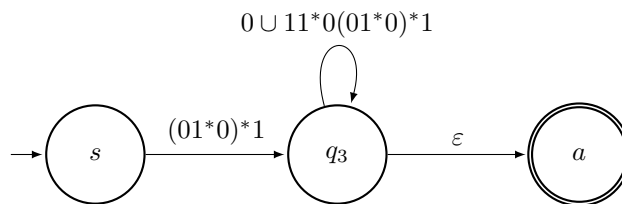


### 3 Transforming Automata [Exam]

The regular expression can be obtained from the finite automaton using the transformation presented in the script on slide 1/85. After ripping out state  $q_2$ , the corresponding GNFA looks like this:



After also removing state  $q_1$ , the GNFA looks as follows.



Eliminating the last state  $q_3$  yields the final solution, which is  $(01^*0)^*1(0 \cup 11^*0(01^*0)^*1)^*$ .

*Note:* Ripping out the interior states in a different order yields a distinct yet equivalent regular expression. The order  $q_3, q_2, q_1$ , for example, results in  $((0 \cup 10^*1)1^*0)^*10^*$ .

### 4 Regular and Context-Free Languages

- a) Sometimes, even simple grammars can produce tricky languages. We can interpret the 1s and 2s of the second production rule as opening and closing brackets. Hence,  $L(G)$  consists of all correct bracket terms where at least one 0 must be in each bracket.

Choose  $w = 1^p 0 2^p \in L(G)$ . Let  $w = xyz$  with  $|xy| \leq p$  and  $|y| \geq 1$  (pumping lemma). Because of  $|xy| \leq p$ ,  $xy$  can only consist of 1s. According to the pumping lemma, we should have  $xy^i z \in L$  for all  $i \geq 0$ . However, by choosing  $i = 0$  we delete at least one 1 and get a word  $w' = 1^{p-|y|} 0 2^p$  with  $|y| \geq 1$ .  $w'$  is not in  $L$  since it has fewer 1s than 2s. This means that  $w$  is not pumpable and hence,  $L(G)$  is not regular.

- b) Since *every* regular language is also context-free, we can choose an arbitrary regular language. For example, we can choose the language  $L = \{0^n 1, n \geq 1\}$  which is clearly regular. A context-free grammar for this language uses only the productions  $S \rightarrow 0S \mid A, A \rightarrow 1$ .

### 5 Context-Free Grammars

- a) An example for a grammar  $G$  producing the language  $L_1$  is  $G = (V, \Sigma, R, S)$  with

$$\begin{aligned}
 V &= \{X, A\}, \\
 \Sigma &= \{0, 1\}, \\
 R &= \left\{ \begin{array}{l} X \rightarrow XAX \mid A, \\ A \rightarrow 0 \mid 1 \end{array} \right\}, \\
 S &= X
 \end{aligned}$$

*Note:* The language is regular!

b) A rather natural grammar generating  $L_2$  uses the following productions:

$$\begin{aligned} S &\rightarrow A1A \\ A &\rightarrow A1 \mid 1A \mid A01 \mid 0A1 \mid 01A \mid A10 \mid 1A0 \mid 10A \mid \varepsilon \end{aligned}$$

Another slightly more complicated solution yielding simpler productions looks as follows:

$$\begin{aligned} S &\rightarrow A1A \\ A &\rightarrow AA \mid 1A0 \mid 0A1 \mid 1 \mid \varepsilon \end{aligned}$$

The idea of both grammars is to first ensure that there is at least one 1 more and then have a production that generates all possible strings with the same number of 0s and 1s or further 1s at arbitrary places.

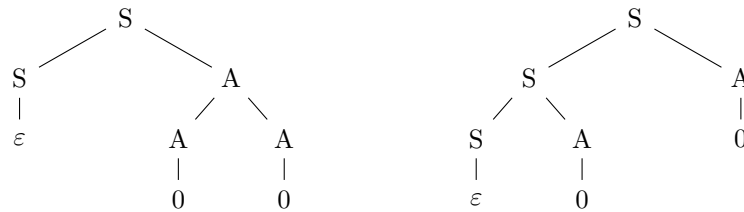
## 6 Ambiguity

a)  $\varepsilon, 0, 00, (), (0), 0(), ()0, 000$

b) It is ambiguous, because the word 00 has two different leftmost derivations.

$$\begin{array}{ll} S \rightarrow SA & S \rightarrow SA \\ \rightarrow A & \rightarrow SAA \\ \rightarrow AA & \rightarrow AA \\ \rightarrow 0A & \rightarrow 0A \\ \rightarrow 00 & \rightarrow 00 \end{array}$$

It can also be seen by taking a look at these two derivation trees that both belong to the word 00:



Because the two derivation trees are structurewise different, the word 00 can be derived ambiguously from  $G$ .

### Ambiguity of Grammars

*Definition:* A string  $s$  is derived *ambiguously* in a context-free grammar  $G$  if it has two or more different leftmost/rightmost derivations (or two structurewise different derivation trees). Grammar  $G$  is *ambiguous* if it generates some string ambiguously.

A *leftmost/rightmost* derivation replaces in every step the leftmost/rightmost variable.

*Example:* The grammar with the productions ' $S \rightarrow S \cdot S \mid S + S \mid a$ ' is ambiguous since the string  $s = a \cdot a + a$  has two different leftmost derivations.

$$\begin{array}{ll} S \rightarrow S \cdot S & S \rightarrow S + S \\ \rightarrow a \cdot S & \rightarrow S \cdot S + S \\ \rightarrow a \cdot S + S & \rightarrow a \cdot S + S \\ \rightarrow a \cdot a + S & \rightarrow a \cdot a + S \\ \rightarrow a \cdot a + a & \rightarrow a \cdot a + a \end{array}$$

Intuitively, the derivation on the left corresponds to the arithmetic expression  $a \cdot (a + a)$  because we first derive a product and then substitute one factor by a sum while the derivation on the right corresponds to  $(a \cdot a) + a$  because we first have a sum and then substitute one summand by a product.

The productions of an equivalent non-ambiguous grammar are  $A \rightarrow S + a \mid S \cdot a \mid a$ .