Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

HS 2013

Prof. R. Wattenhofer / K.-T. Foerster, T. Langner, J. Seidel

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.

a) Language L_1 can be shown to be non-regular using the pumping lemma. Assume for contradiction that L_1 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_1 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 $xy^iz \in L_1$. 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.

(1) 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. $1\,0^p\,1^p$ for $y = 01$).

(2) 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_1. Therefore, L_1 cannot be regular and we have a contradiction.

b) With the adapted language L_2, the proof of non-regularity is much more tricky! Specifically, non-regularity of L_2 cannot be proven using the pumping lemma, because any word in L_2 can actually be pumped! Consider for instance a word w of the form 0110^p1^p. In this case, we can split w into the three parts $x = 0, y = 11, z = 0^p1^p$, which is in accordance with the rules of the pumping lemma. It can be seen, however, that any word xy^iz is also in L_2! That is, the language L_2 can be pumped and yet, it is not regular as shown below.

Assume for contradiction that there exists a finite automaton A which accepts the language L_2. Every word that starts with the input-sequence 0110 is only accepted if the remainder of the word has the form $0^{c-1}1^c$ for some integer $c > 0$. Let q_1 be the state reached after the input 0110. Given the automaton A, we can construct a regular automaton A' that is equivalent to A with the only difference that its initial state is q_1. By the definition of A, this adapted finite automaton A' accepts all words of the form $0^{c-1}d^c$. However, as shown on slide 1/95 of the script, the language $0^{c-1}d^c$ is not regular. Hence, A' and thus A cannot be finite automata. Because there exists a finite automaton for every regular language, it follows that L_2 cannot be regular. Language L_2 shows that while every regular language can be pumped according to the pumping lemma, there are also non-regular languages that can be pumped.

Variant: We can alternatively use the fact that if two languages L and L' are regular, the language defined by the intersection of the two languages $L \cap L'$ is regular as well (cf. p. 1/41). Consider the regular language $L_3 = \{w \in 0110^*1^*\}$. Notice that the intersection of L_3 with $L_2 = \{0^a1^b0^c1^d \mid a, b, c, d \geq 0 \text{ and if } a = 1 \text{ and } b = 2 \text{ then } c = d\}$ contains exactly all words $w \in \{0110^n1^n \mid n \geq 0\}$. This, however, is the exact language L_1 we proved not to be regular in the first part of this exercise. If we assume L_2 to be regular, L_1 must be regular as well, since $L_1 = L_2 \cap L_3$. This is a contradiction. Thus L_2 cannot be regular.

Be Careful!

The argumentation above is based on the closure properties of regular languages and only works in the direction presented. That is, for an operator $\diamond \in \{\cup, \cap, \bullet\}$, we have:

If L_1 and L_2 are regular, then $L = L_1 \diamond L_2$ is also regular.

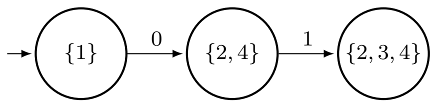
If either L_1 or L_2 or both are non-regular, we cannot deduce the non-regularity of L or vice-versa. Moreover, L being regular does not imply that L_1 and L_2 are regular as well. This may sound counter-intuitive which is why we give examples for the three operators.

- $L = L_1 \cup L_2$: Let L_1 be any non-regular language and L_2 its complement. Then $L = \Sigma^*$ is regular.
- $L = L_1 \cap L_2$: Let L_1 be any non-regular language and L_2 its complement. Then $L = \emptyset$ is regular.
- $L = L_1 \bullet L_2$: Let $L_1 = \{a^*\}$ (a regular language) and $L_2 = \{a^p \mid p \text{ is prime}\}$ (a non-regular language) then $L = \{aaa^*\}$ is regular.

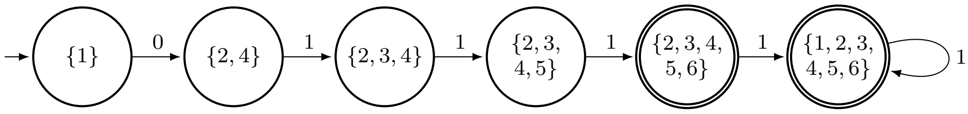
Hence, to prove that a language L_x is non-regular, you assume it to be regular for contradiction. Then you combine it with a *regular* language L_r to obtain a language $L = L_x \diamond L_r$. If L is non-regular, L_x could not have been regular either.

2 Deterministic Finite Automata [Exam]

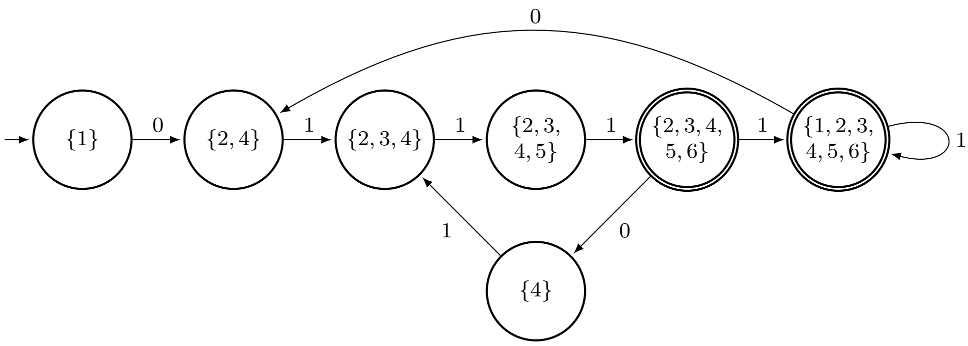
We could use the systematic transformation scheme presented in the lecture (slide 1/75). Considering the large number of states, however, this will easily lead to an explosion of states in the derandomized automaton. Hence, we build the deterministic finite automaton in a step-wise manner, only creating 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.



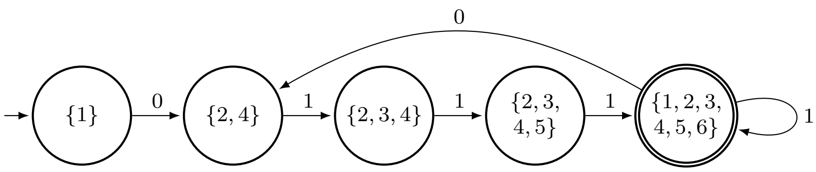
In any of the states 2, 3, and 4, only a 1 is accepted. Assume that the automaton is currently in state 2, this 1 can lead to states $\{2, 3, 4\}$ when including all ε-transitions. When in state 3, the 1 leads to states $\{2, 3, 4, 5\}$ and finally, when being in state 4, the reachable states given a 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.



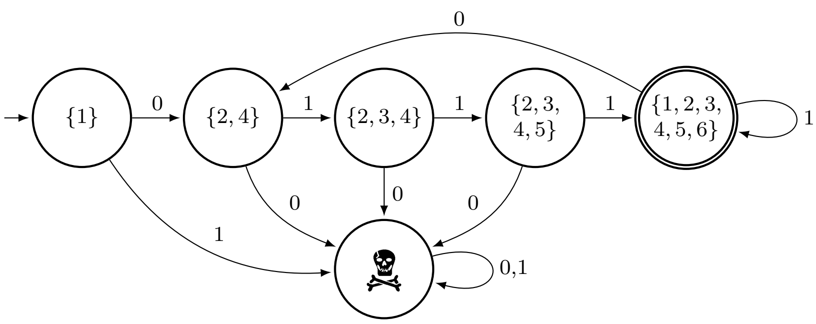
What happens if a 0 occurs in the input? This is feasible only when the deterministic state includes either state 1 or state 6. In state $\{2,3,4,5,6\}$, a 0 necessarily leads to state $\{4\}$, whereas in state $\{1,2,3,4,5,6\}$ a 0 leads to state $\{2,4\}$. In both of these states, the only acceptable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

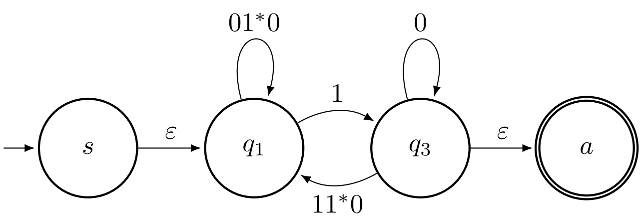


This is not a DFA yet, because the crash state is still missing. The final deterministic automaton looks like this:

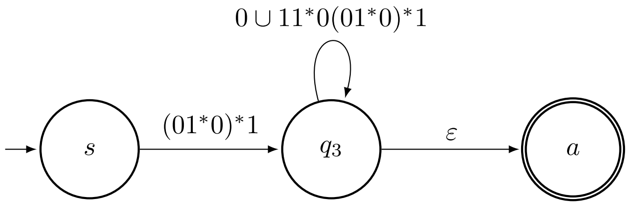


3 Transforming Automata [Exam]

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 $w = 1^p02^p \in L(G)$. 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 $xy^iz \in L$ for all $i \geq 0$. However, by choosing $i = 0$ we delete at least one 1 and get a word $w' = 1^{p-|y|}02^p$ 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 $L = \{0^n1, n \geq 1\}$ which is clearly regular. A context-free grammar for this language uses only the production $S \to 0S \mid 1$.

5 Context-Free Grammars

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

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

Note: The language is regular!

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

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

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

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

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.

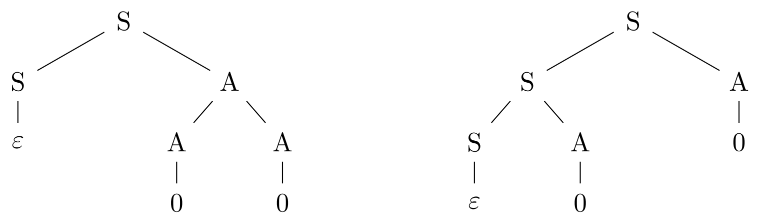
6 Pushdown Automata

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

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

$$\begin{aligned} S &\to SA \\ &\to A \\ &\to AA \\ &\to 0A \\ &\to 00 \end{aligned} \qquad\qquad \begin{aligned} S &\to SA \\ &\to SAA \\ &\to AA \\ &\to 0A \\ &\to 00 \end{aligned}$$

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.

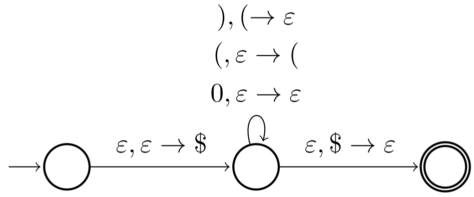
Example: The grammar with the productions '$S \to S \cdot S \mid S + S \mid a$' is ambiguous since the string $s = a \cdot a + a$ has two different leftmost derivations.

$$\begin{aligned} S &\to S \cdot S \\ &\to a \cdot S \\ &\to a \cdot S + S \\ &\to a \cdot a + S \\ &\to a \cdot a + a \end{aligned} \qquad\qquad \begin{aligned} S &\to S + S \\ &\to S \cdot S + S \\ &\to a \cdot S + S \\ &\to a \cdot a + S \\ &\to a \cdot a + a \end{aligned}$$

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 $A \to S + a \mid S \cdot a \mid a$.

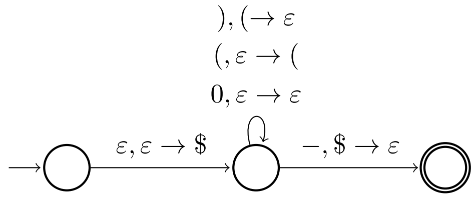
c) A simple non-deterministic PDA for $L(G)$ looks as follows:



> **Deterministic PDAs**
>
> A push-down automaton M is *deterministic* iff in each state, there is exactly one successor state for every combination $(a, b) \in \Sigma \times \Gamma$ where Σ is the string input alphabet and Γ is the stack alphabet. Note that if a state q has only one outgoing transition '$\varepsilon, \varepsilon \to \$$' the PDA is still deterministic since there is no ambiguity of what the successor state of q will be. If a state q, however, has two outgoing transitions, '$a, \varepsilon \to x$' and '$\varepsilon, b \to y$' leading into different states, it is unclear which transition the system should take if the string input in state q is 'a' and the top element on the stack is 'b'. A PDA containing such ambiguous transisitions is *not* deterministic.
>
> Unlike in deterministic finite automata, we take the liberty of omitting transitions leading to an (imaginary) fail state as well as the fail state itself when drawing deterministic PDAs.

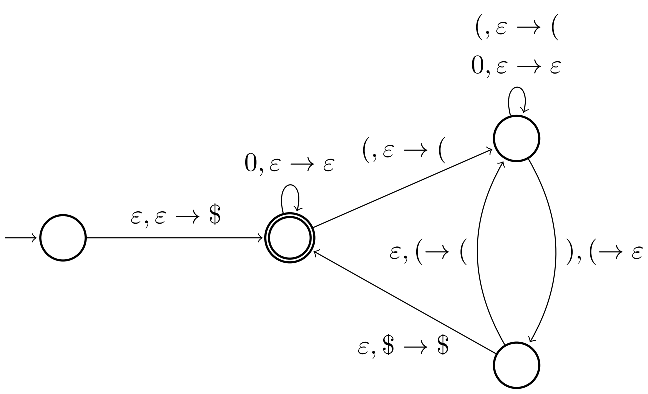
Considering this, the PDA given above is not deterministic: From the middle state, there are two transitions '$(, \varepsilon \to ($' and '$\varepsilon, \$ \to \varepsilon$', such that we do not know which one to take if we read a '(' while the top element on the stack is '\$'. We can overcome this problem in different ways.

If we assume that our PDA recognizes the end of the input string (denoted by '$-$'), it is easy to transform the non-deterministic PDA above into a deterministic one:



If we assume that the PDA is not able to determine the end of the input, it is not that easy to derive the deterministic PDA from the non-deterministic one.

An example of a deterministic PDA accepting $L(G)$ is the following:



The deterministic PDA using as few states as possible is the following: