Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems Group (NSG)

HS 2016

Prof. L. Vanbever & A. El-Hassany, & M. Apostolaki.
based on Prof. R. Wattenhoffer's material

Discrete Event Systems

Solution to Exercise Sheet 3

1 Pumping Lemma Revisited

a) Let us assume that L is regular and show that this results in a contradiction.

We have seen that any regular language fulfills the pumping lemma. This means, there exists a number p, such that every word $w \in L$ with $|w| \geq p$ can be written as $w = xyz$ with $|xy| \leq p$ and $|y| \geq 1$, such that $xy^iz \in L$ for all $i \geq 0$.

In order to obtain the contradiction, we need to find at least one word $w \in L$ with $|w| \geq p$ that does not adhere to the above proposition. We choose $w = xyz = 1^{p^2}$ and consider the case $i = 2$ for which the Pumping Lemma claims $w' = xy^2z \in L$.

We can relate the lengths of $w = xyz$ and $w' = xy^2z$ as follows.

$$p^2 = |w| = |xyz| < |w'| = |xy^2z| \leq p^2 + p < p^2 + 2p + 1 = (p + 1)^2$$

So we have $p^2 < |w'| < (p+1)^2$ which implies that $|w'|$ cannot be a square number since it lies between two consecutive square numbers. Therefore, $w' \notin L$ and hence, L cannot be regular.

b) Consider the alphabet $\Sigma = \{a_1, a_2, ..., a_n\}$ and the language $L = \bigcup_{i-1}^{n} a_i^*$. The language is regular, as it is the union of regular languages, and the smallest possible pumping number p for L is 1. But any DFA needs at least $n+1$ states to distinguish the n different characters of the alphabet. Thus, for the DFA, we cannot deduce any information from p about the minimum number of states.

The same argument holds for the NFA.

2 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.

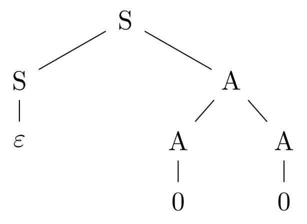
3 Pushdown Automata

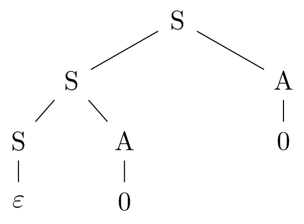
a) $\varepsilon, 0, 00, (), (0), 0(), ()0, 000$

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.

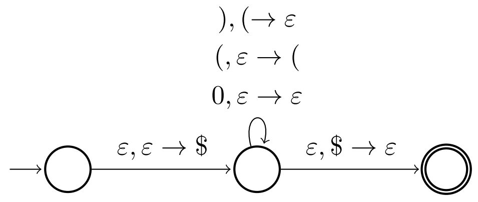
Example: The grammar with the productions '$S \to S \cdot S \mid S + S \mid a$' is ambiguous since the string $s = a \cdot a + a$ has two different leftmost derivations.

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

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:



$), (\rightarrow \varepsilon$
$(, \varepsilon \rightarrow ($
$0, \varepsilon \rightarrow \varepsilon$
$\varepsilon, \varepsilon \rightarrow \$$
$\varepsilon, \$ \rightarrow \varepsilon$

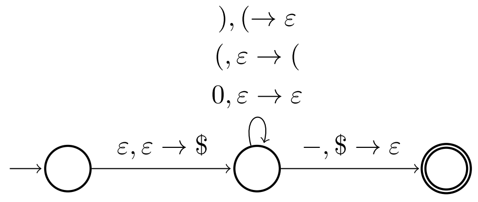
Deterministic PDAs

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 '$\varepsilon, \varepsilon \rightarrow \$$' 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, '$a, \varepsilon \rightarrow x$' and '$\varepsilon, b \rightarrow y$' 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 '$(, \varepsilon \rightarrow ($' and '$\varepsilon, \$ \rightarrow \varepsilon$', 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.

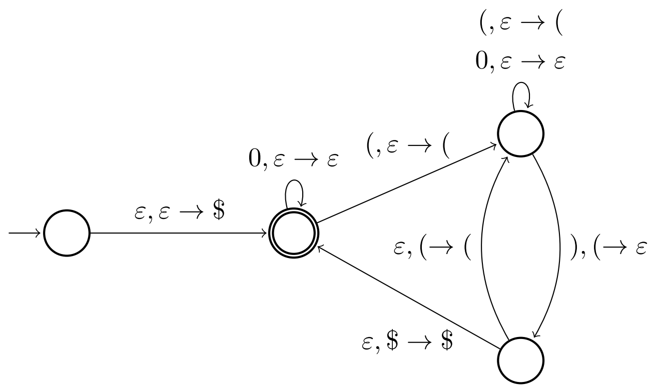
If we assume that our PDA recognizes the end of the input string (denoted by '$-$'), it is easy to transform the non-deterministic PDA above into a deterministic one:



$), (\rightarrow \varepsilon$
$(, \varepsilon \rightarrow ($
$0, \varepsilon \rightarrow \varepsilon$
$\varepsilon, \varepsilon \rightarrow \$$
$-, \$ \rightarrow \varepsilon$

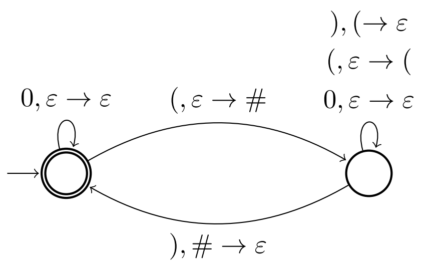
If we assume that the PDA is not able to determine the end of the input, it is not that easy to derive the deterministic PDA from the non-deterministic one.

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



$(, \varepsilon \rightarrow ($
$0, \varepsilon \rightarrow \varepsilon$
$0, \varepsilon \rightarrow \varepsilon$
$(, \varepsilon \rightarrow ($
$\varepsilon, \varepsilon \rightarrow \$$
$\varepsilon, (\rightarrow ($
$), (\rightarrow \varepsilon$
$\varepsilon, \$ \rightarrow \$$

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



4 Context Free or Not?

a) For reasons of brevity, we only give the productions of the grammar.

First, we create an equal number of symbols for w and z using rule (2), and then an equal number of symbols for x and y using rule (3).

$$\begin{aligned}
S &\to A && (1)\\
A &\to YAY \mid \#B\# && (2)\\
B &\to YBY \mid \# && (3)\\
Y &\to a \mid b && (4)
\end{aligned}$$

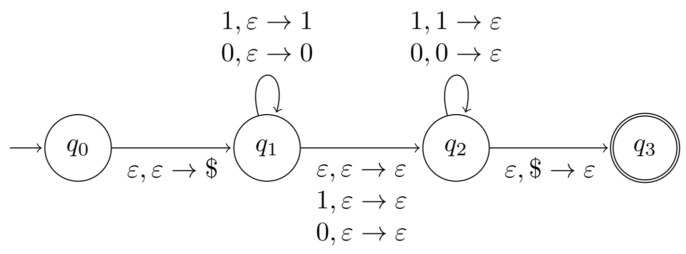
b) If $|w| = |y|$ and $|x| = |z|$, the resulting language is not context free, thus a CFG does not exist. This can be seen using the tandem pumping lemma as follows.

Let the word considered be $s = a^p\#a^p\#a^p\#a^p \in L$ with $|s| = 4p + 3 \geq p$. For any division $s = defgh$ with $|eg| \geq 1$ and $|efg| \leq p$, the pumpable regions e and g can never consist of boths as from w and y or both x and z because of the condition $|efg| \leq p$. Hence, any pumping would inevitably only modify the number of as in one part thereby creating a word $s' \notin L$. Therefore, L cannot be context free.

5 Push Down Automata

a) This PDA should recognize all palindromes. However, we don't know where the middle of the word to recognize is. Therefore, we have to construct a non-deterministic automaton that decides itself when the middle has been reached.

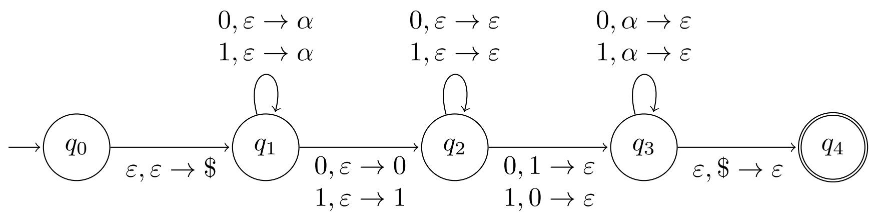
Note that we need to support words of even and odd length. Words of even length have a counterpart for each letter. However, the center letter of an odd word has no counterpart.


$1, \varepsilon \to 1$
$0, \varepsilon \to 0$
$1, 1 \to \varepsilon$
$0, 0 \to \varepsilon$
q_0
q_1
q_2
q_3
$\varepsilon, \varepsilon \to \$$
$\varepsilon, \varepsilon \to \varepsilon$
$1, \varepsilon \to \varepsilon$
$0, \varepsilon \to \varepsilon$
$\varepsilon, \$ \to \varepsilon$

b) Consider the word w to be an array of symbols. If $w \in L$, there is at least one offset c, such that $w[c] \neq w[|w| - c]$. That is, there are two symbols x and y in w s.t. $x \neq y$ and the distance of x from the start of w equals the distance of y from the end of w.

The PDA reads $c - 1$ symbols, and stores a token α on the stack for each read symbol. Then, it reads the c-th symbol, and puts the symbol onto the stack. Afterwards, the PDA allows to read arbitrarily many symbols from the input, and does not modify the stack. Then, when only c symbols are left on the input stream, the PDA requires that the symbol on the stack must differ from the one on the input. Finally, the PDA reads the remaining $c-1$ symbols and accepts if the stack is empty.

Note that this is again a non-deterministic PDA, as we do not know the value of c.


$0, \varepsilon \to \alpha$
$1, \varepsilon \to \alpha$
$0, \varepsilon \to \varepsilon$
$1, \varepsilon \to \varepsilon$
$0, \alpha \to \varepsilon$
$1, \alpha \to \varepsilon$
q_0
$\varepsilon, \varepsilon \to \$$
q_1
$0, \varepsilon \to 0$
$1, \varepsilon \to 1$
q_2
$0, 1 \to \varepsilon$
$1, 0 \to \varepsilon$
q_3
$\varepsilon, \$ \to \varepsilon$
q_4

6 Counter Automaton

a) A counter automaton is basically a finite automaton augmented by a counter. For every regular language $L \in \mathcal{L}_{reg}$, there is a finite automaton A which recognizes L. We can construct a counter automaton C for recognizing L by simply taking over the states and transitions of A and *not* using the counter at all. Clearly C accepts L. This holds for every regular language and therefore, $\mathcal{L}_{reg} \subseteq \mathcal{L}_{count}$.

b) Consider the language L of all strings over the alphabet $\Sigma = \{0, 1\}$ with an equal number of 0s and 1s. We can construct a counter automaton with a single state q that increments/decrements its counter whenever the input is a 0/1. If the value of the counter is equal to 0, it accepts the string. Hence, L is in $\mathcal{L}_{count}$. On the other hand, it can be proven (using the pumping lemma) that L is not in $\mathcal{L}_{reg}$ and it therefore follows $\mathcal{L}_{count} \nsubseteq \mathcal{L}_{reg}$.

Some languages where the (non-finite) frequency of one or several symbols depends on the frequency of other symbols can be recognized by counter automata. Such languages cannot be recognized by finite automata.

c) First, we show that a pushdown automaton can simulate a counter automaton. Hence, PDAs are at least as powerful as CAs! The simulation of a given CA works as follows. We construct a PDA which has exactly the same states as the CA. The transitions also remain between the same pairs of states, but instead of operating on an INC/DEC register, we have to use a stack. Concretely, we store the state of the counter on the stack by pushing '+' and '−' on the stack. For instance, a counter value '3' is represented by three '+' on the stack, and similarly a value '−5' by five '−'. Therefore, when the CA checks whether the counter equals 0, the PDA can check whether its stack is empty.

In the following, we give just one example of how the transitions have to be transformed. Assume a transition of the counter automaton which, on reading a symbol s, increments the counter—independently of the counter value. For the PDA, we can simulate this behavior with three transitions: On reading s and if the top element of the stack is '−', a minus is popped; if the top element is a '+', another '+' is pushed; and if the stack is empty, also a '+' is pushed.

Hence, we have shown that the PDA is at least as powerful as the CA, and it remains to investigate whether both CA and PDA are equivalent, or whether a PDA is stronger. Although it is known that the PDA is actually more powerful, the proof is difficult: There is no pumping lemma for CAs for example such that we can prove that a given context-free language cannot be accepted by a CA. However, of course, if you have tackled this issue, we are eager to know your solution... :-)