Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Networked Systems Group (NSG)

HS 2017

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

Discrete Event Systems

Solution to Exercise Sheet 4

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 $xy^i z \in L$ 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 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).

$$S \to A \tag{1}$$
$$A \to YAY \mid \#B\# \tag{2}$$
$$B \to YBY \mid \# \tag{3}$$
$$Y \to a \mid b \tag{4}$$

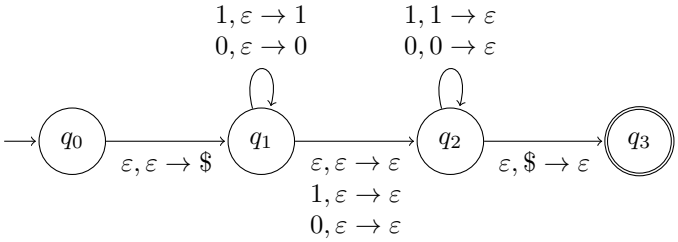
b) If $|w| = |y|$ and $|x| = |z|$, the resulting language is not context free, thus a CFG does not exist. This can be seen using the tandem pumping lemma as follows.

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.

3 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.

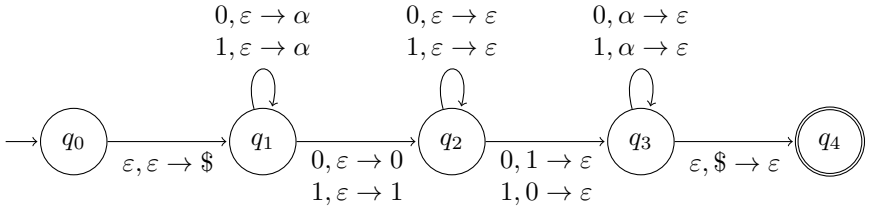
Note that we need to support words of even and odd length. Words of even length have a counterpart for each letter. However, the center letter of an odd word has no counterpart.



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.



4 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... :-)

5 Designing Turing Machines

The proposed Turing machine decrements the value of a until $a = 0$. In each step, it adds a '1' to the output:

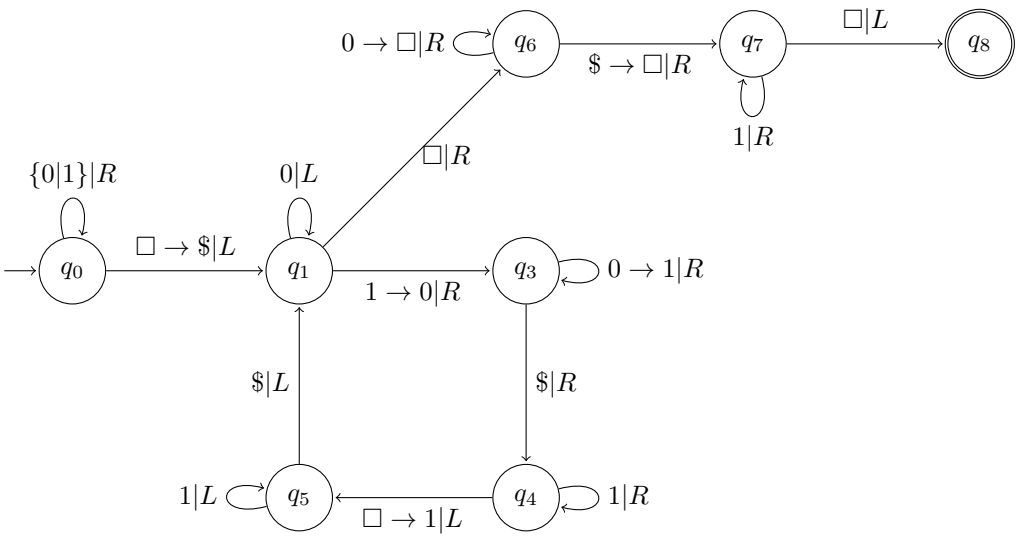
1. Move the TM head to the right of a and place a $ sign. We will use this marker to return to the LSB of a.

2. Look at the LSB of a. If it is '1', we change it to 0 (transition between q_1 and q_3) and move to the right. Then, we continue moving to the right until we hit a □, which is changed to a '1' (transition q_4 to q_5). Finally, we move back to the LSB of a.

3. If the LSB of a is '0', we search for the first '1' in a from the right (loop on q_1 and transition from q_1 to q_3).

3.1 If we find a '1', we change it to '0'. While moving back to the $ symbol, we change all '0' to '1' (self-loop on q_3). Then, we proceed as in point 2 after passing the $ symbol.

3.2 If we don't find a '1' in a at all (transition q_1 to q_6), we start the cleanup procedure: Remove all 0 on the right of the $ symbol, and finally remove the $ symbol itself and move to the right of u.



6 An Unsolvable Problem

a) It is surprisingly easy to prove that your boss is demanding too much. Assume a function `halt(P: Program): boolean` which takes a program `P` as a parameter and returns a boolean value denoting whether `P` terminates or not.

Now consider the following program `X` which calls the `halt()` function with itself as an argument just to do the contrary:

```
function X() {
  if (halt(X))
    while(true);
  else
   return;
}
```

Obviously, if `halt(X)` is true `X` will loop forever, and vice versa.

b) If the simulation stops we can definitively decide that the program does not contain an endless loop. However, while the simulation is still running, we do not know whether it will finish in the next two seconds or run forever. Put differently: There is no upper bound on the execution time of the simulation after which we can be sure that the program contains an endless loop.

c) As we have seen, it is not possible to predict whether a general program terminates or not. However, under certain constraints we can solve the halting problem all the same. For example, consider a restricted language with only one form of a loop (no recursion etc.):

```
for (init; end; inc) {...}
```

where `init`, `end` and `inc` are constants in $\mathbb{Z}$. The loop starts with the value `init` and adds `inc` to `init` in every round until this sum exceeds `end` if `end` > 0 or until it falls below `end` if `end` < 0. Obviously, there is a simple way to decide whether a program written in this language terminates: For every loop, we check whether sgn(`inc`) = sgn(`end`), where sgn$(\cdot)$ is the algebraic sign. If not, the program contains an endless loop (unless `init` itself already fulfills the termination criterion which is also easy to verify).