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

HS 2018

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

Discrete Event Systems

Solution to Exercise Sheet 5

1 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.

2 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.

A counter automaton can recognize languages where the (non-finite) frequency of one symbol depends on the frequency of the other symbols. Such languages cannot be recognized by finite automatons.

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... :-)

3 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).