Theory of Computer Science

G. Röger
Spring Term 2020

University of Basel
Computer Science

Exercise Sheet 8 — Solutions

Exercise 8.1 (Automata; 2 marks)

Write a summary of about the different types of automata covered in Chapter C. You do not have to explain what a DFA/NFA/PDA/DTM/NTM is or how it is defined. Instead, your summary should focus on the differences of the automata and how those differences influence the complexity of the languages that can be accepted with them. For example, in what way can the automata store information while reading the input, and how much information is necessary to recognize certain languages? *A good answer can be written in about half a page in LaTeX.*

Solution:

The following is one possible answer but since this is an open question, other answers are possible.

DFAs, NFAs, and PDAs can only read the input word once from left to right. That means that if the language has requirements on different parts of the word, the automaton has to "remember" some information about the word. For example, in the language $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ an automaton has to remember the number of `a`s to later compare them the number of `b`s. DFAs and NFAs can only store information in their current state. Since the number of states is finite, this means that they can only store a finite amount of data. Regular languages, for example, the language of all words that end in `b` can be recognized by a DFA because they only require storing a finite amount of data (in this case, whether the last symbol we read was a `b`). The pumping lemma is build around this observation: if a language is regular and can be accepted by a DFA/NFA, then it has to have repeatable parts because the DFA/NFA can only store a finite amount of data.
PDAs can store additional information on a stack. Compared to DFAs/NFAs this makes them more powerful because the stack does not have a limit on the number of entries. Context-free languages such as $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ can be recognized by PDAs: while reading the `a`s, the PDA can "count" them by adding elements to the stack. When reading the `b`s it can then compare their number to the number of symbols on the stack by removing one symbol from the stack for every `b` read. However, PDAs can only access their stack from the top, and they consume the elements they read from it. For example, it is not possible to peek at the bottom of the stack without removing all other elements. This means that non-context-free languages like $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ can not be accepted by PDAs.
Turing machines are even more powerful than PDAs because they can store an arbitrary amount of information on their tape and can access any part of this information at any time without consuming it. For every language that can be generated by a grammar there is a Turing machine that accepts it, and it is the ability to store and access an arbitrary amount of information that makes this possible. Later in the course, we will see that Turing machines still have their limitations.

Exercise 8.2 (Deterministic Turing machines; 2+2+2+2 marks)

In this exercise you should implement a simulator for a deterministic Turing machine.
You can find a basic framework for this on the lecture homepage. Implement the given methods and chose appropriate private fields for the classes. Only implement the places marked with `// TODO` in the code; do not add any additional methods, classes or public fields.

(a) Implement the following methods in the class `Tape` that represents the Turing machine's tape which is infinite in both directions and the position of the read/write head:

The constructor `Tape(word, blank)` accepts the word that is on the tape initially and the blank symbol that should be used. The read/write head initially is over the first symbol of the word.

The method `read()` should return the symbol under the read/write head.

The method `write(symbol)` should write `symbol` on the tape to the current position of the read/write head.

The methods `moveLeft()` and `moveRight()` should move the head one position to the left or right respectively.

The method `dumpAlpha()` should print out the part of the tape that is left of the current position of the read/write head (excluding the current position).

The method `dumpBeta()` should print out the part of the tape that is right of the current position of the read/write head (including the current position).

The method `usedSpace()` should return the size (the number of tape positions) of the tape that was used so far.

(b) Implement the constructor `TuringMachine(Q, Sigma, Gamma, delta, q0, blank, E)` where the parameters correspond to the normal elements of a Turing maschine. During construction make sure that the Turing machine is specified correctly: is `delta` a total function? Is `Sigma` a subset of `Gamma`? Is the `blank` in `Gamma` but not in `Sigma`? Is `E` a subset of `Q`? Throw an `InvalidSpecificationException`, if the specification is incorrect.

(c) Implement the remaining methods in the class `TuringMachine`:

`initialize(word)` should initialize the Turing machine, i.e., create the initial configuration. Make sure that only symbols from the input alphabet occur in `word`. Throw a `InvalidSpecificationException` if this is not the case.

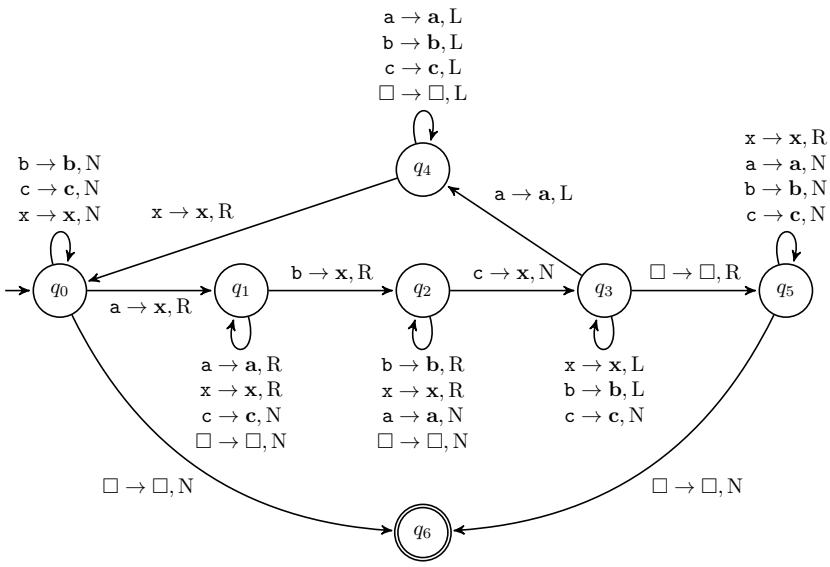
`step()` should execute one step of the Turing machine. Throw a `RuntimeException` if the machine is not initialized or is in a final state.

The method `dumpConfiguration()` should print out the current configuration of the Turing machine.

The method `dumpStatistics()` should print out how many steps where simulated so far and how much space was used on the tape so far.

`run(maxSteps)` should run the Turing machine. The machine should only stop if it reaches a final state or more than `maxSteps` steps were executed. This is useful to test the machine on a word where the machine does not terminate. The method `run()` calls `run(Integer.MAX_VALUE)` to simulate an unrestricted run. Call `dumpConfiguration()` at the start of the simulation and after every step, and call `dumpStatistics()` at the end of the simulation.

(d) Consider the following turing machine.



Implement the `main` method in the class `SimulateTM` that should create an object of the class `TuringMachine` representing the Turing machine above. Test different inputs for the machine, containing at least:

- the empty word, and
- two words that are not in the language accepted by the Turing machine (but not the empty word or *aabbcc*), and
- two words that are not in the language accepted by the Turing machine.

Print out the configuration in every step of the simulation. For words that are not in the language, use a small value for `maxSteps` that already shows that the Turing machine will never terminate. For words that are in the language, use `run()` without a parameter.

You can check your implementation with the input *aabbcc*: the Turing machine requires 29 steps and 8 tape cells for this input.