Chapter 4

Introduction to Finite Automata and Regular Expressions

The first model of a computer that we are going to introduce is the *finite automaton* (*finite state machine* or *finite state automaton*[1]). This is the simplest and, in a sense, the least powerful model that we will consider. In some ways, it is so limited that it is obviously not a good model of a computer, as it lacks some of the features that even the most rudimentary computers have (such as writable memory). However, studying finite automata is still very valuable for a number of reasons. Firstly, it represents an excellent introduction to the formal study of computer systems, being simple enough that it is easily understandable, but, at the same time, strict and formal enough that it requires careful logical reasoning. Secondly, while complete computer systems cannot (feasibly) be modeled using finite automata, some important parts of them can be and the study of finite automata allows us to construct and formally prove some properties of such subsystems.

The finite automaton model can be seen as a machine that reads an input step-by-step and, at each step, potentially changes its internal state. The input that the automaton receives is a string of symbols. Thus, the finite automaton model will have several important features:

- it is always in one or more of a finite number of states;
- it receives as an input a string and reads that string symbol-by-symbol;
- it can change the state at each step (where one step is reading of one symbol) and even between the steps;
- the states in which the automaton is depend solely on the sequence of input symbols read so far;
- it has no internal memory as such, but it is still able to "remember" information (via states);
- it produces no output - it only decides whether the input string is *accepted* or *rejected*

[1] Note that the plural of *automaton* is *automata*.

Some of the questions that we typically deal with in the automata theory are

1. Does a given automaton M accept the given string w?
2. What strings does a given automaton M accept? In other words, what is the language accepted by M?
3. Can we, for a given language L, construct a finite automaton that accepts that language?

We are going to consider several specific finite automata models, starting from the simplest one, which is Deterministic Finite Automata (Chapter 5). These automata can be in only one state at a time and cannot spontaneously move between states without processing any input symbol. Afterwards, we will consider Nondeterministic Finite Automata (Chapter 6), which can be in more than one state at a time, but cannot move spontaneously move between states. Finally, we are going to consider Nondeterministic Finite Automata With ϵ-transitions, which can be in more than one state at a time and can move spontaneously between the states (Chapter **??**). Perhaps surprisingly, we will see that all of these kinds of automata have the same power, in that there is no language that is accepted by an automaton of one kind that cannot be also accepted by some automaton of another kind. Thus, nondeterminism and spontaneous transitions do not increase the power of an automata, but can make the automaton to accept some language easier to design and more concise.

Another important concept that we are going to introduce is a concept of regular expressions (Chapter **??**), which are heavily used in programming and can signifcantly ease the tasks of text processing and manipulation. Regular expressions are expressions built using a small number of simple ruless. Each regular expression describes a language. Another surprising result that we are going to prove (Chapter **??**) is that, while the regular expressions appear to be a completely different concept than finite automata, the languages that they define are exactly the same languages accepted by finite automata. That is, if a language is described by some regular expression, it is also accepted by some finite automaton, and vice versa. This is a very important result as it represents the first link between the languages and the automata that was ever established. We will finish this chapter by looking at one particularly effective method of proving that a language is *not* regular - the Pumping Lemma (Chapter **??**). Using this lemma, we will prove that some reasonably simple languages are not regular and that, therefore, there is no finite automata (deterministic, non-deterministic or with ϵ-transitions) that can accept these languages. This shows the limits of finite automata as models of computers and give motivation for considering more complex (but also massively more powerful) models such as Turing Machines.

Chapter 5

Deterministic Finite Automata

5.1 Introduction and Motivating Example

The first kind of finite automata that we are going to introduce is the *deterministic finite automata* (or DFAs, for short). With respect to the overall list of features of the finite automata, DFAs have two important restrictions:

1. A DFA is always in *exactly one* state.
2. A DFA can change its state only upon processing an input symbol (and not spontaneously).

Before we formally define a deterministic finite automaton, let su first consider one simple example.

Example 5.1: Parking Ticket Machine

Let us design a parking ticket machine as a DFA. The machine should accept 50p, £1 and £2 coins and should require a payment of £2 (no change given) to issue a whole day parking ticket. *Input* to the ticket machine is a sequence of coins put in by a machine user, where each coin belongs to a set $\{50p, £1, £2\}$. There is no output - after the sequence of coins is inserted, the machine either *accepts* the sequence (and issues a ticket) or, otherwise, *rejects* it (by simply not issuing a ticket).

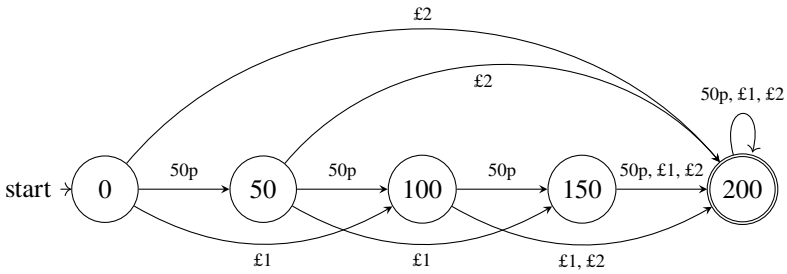
We now need to define the states of our DFA and the way of moving between the states depending on the coin inserted. What should the states of our automaton be? Intuitively, we know that the machine somehow needs to remember the amount of money that is put in. However, we said that a DFA has no memory, so we need some way of remembering this information other than storing it into memory. This is what we will use the states for - each different state will correspond to a different amount of money put in. This is in perfect agreement with one of the other features of the DFA, namely that it can change its state based on the input symbol that it receives. As we put in more coins, the total amount of money put in increases and therefore the state of the automaton changes (until we put in enough money for a ticket, after which putting in more money does not really change anything). The set of states that we will have has to account for each possible amount of money put in. It is easy to see that we need to have five different states:

1. no money has been put in;
2. 50p has been put in;
3. £1 has been put in;
4. £1.50 has been put in;
5. £2 or more has been put in

As observed before, we do not need separate states for amounts greater than £2, as they don't change anything in terms of the accept/reject outcome; regardless of whether we put in £2 or £50, the outcome is the same - the sequence of coins will be accepted and the ticket will be issued. Our DFA will, then, have the above five states, which we can name according to the amount of pennies put in - 0, 50, 100, 150 and 200.

Now that we have our states, we need to determine how our DFA moves between these states as a result of processing input. The automaton will start in the state 0. If a 50p coin is inserted, it will move to the state 50. If a £1 coin is inserted, it will move to the state 150. Finally, if another 50p (or, indeed, £1 or £2) coin is inserted, it will move to the state 200. It is very easy to conclude the similar for all of the other states. For example, if the automaton is in the state 150, receiving any input will move it to the state 200, and if the automaton is in the state 200, receiving any input will make it remain in the state 200. Remember that the state does not need to change as a response to an input read.

As a final step in constructing our DFA, we need to determine what the starting state of our automaton will be, and what will the *accepting* states be - that is, in which states the input sequence of coins should be accepted and the ticket issued, and in which it should be rejected and no ticket issued. It is easy to see that the state 0 is the starting state, while the state 200 is the only accepting state. If, after processing the full sequence of coins, the automaton is in this state, then enough money has been put in and a ticket should be issued. In any other state, not enough money has been put in, and therefore if the input sequence ends when the automaton is in one of these states, the ticket should not be issued. Our DFA can be represented by a diagram below



In this diagram, the accepting state 200 is surrounded by a double circle to distinguish it from the non-accepting states, and the state 0 has an arrow labelled 'start' that leads to it, distinguishing it as a state in which the automaton starts. Note that we can consider an input to our DFA as a string over the alphabet $\{50\text{p}, £1, £2\}$. $50p$, 1 and 2 are the symbols. To make it easier to comprehend, we can use the f, p and t as symbols for 50p, £1 and £2, respectivelly. Then, for example, inserting two $50p$ coins, followed by inserting a 1 coin, would correspond to feeding a ffp string as an input to our DFA.

It is easy to see how, for example, the states of the DFA change when the sequence of coins $(50\text{p}, £1, £1)$ is read as an input. It is easy to see that the DFA moves from the state 0 to the state 50 (upon reading the 50p coin), then to the state 150 (upon reading the £1 coin) and finally to the state 200 (upon reading another £1 coin). In this state the DFA ends processing of the input, as all the coins have been processed, and therefore it will accept the given sequence of coins and issue the ticket.

Let us consider for a moment what we needed in order to describe this DFA completely. We can see that we needed to decide on five things:

- What states the DFA has, together with their names (so that we can distinguish between the states)?
- What input symbols are allowed in the input strings?
- How we move between the states when we receive an input symbol?
- In which state the DFA starts?
- In which states it needs to end the computation in order to accept the input?

These five things lead us to the formal definition of a DFA. But before that, observe that the automaton in our example can be only in one state at each step and that it can move between steps only as a response to reading an input. Therefore, the automaton that we designed will, indeed, be a deterministic finite automaton.

5.2 Formal Definition of a Deterministic Finite Automaton

We are now ready to give a formal definition of a deterministic finite automaton.

Definition 5.1: Deterministic Finite Automaton (DFA)

Deterministic Finite Automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of *states*
- Σ is a finite *alphabet* (set of possible input symbols)
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $q_0 \in Q$ is the *start state*
- $F \subset Q$ is the set of *accepting states*

Let us now analyse this definition part-by-part.

States. Firstly, the set Q is the set of states. We can see this simply as the set of labels we have choosen for the states of our DFA. We have some freedom here in that renaming the states does not in any way impact the operation of the automaton, as long as we are consistent. In our example above, we could have easily named the states A, B, C, D, E, corresponding to the old names $0, 50, 100, 150, 200$. As long as the automaton moves between the states with new names exactly in the same way as it does between the states with the corresponding old names (e.g. it moves from A to B on input 50p), and as long we are consistent with the starting and the accepting state (A and E, respectivelly), the outcome of processing any input sequence will be the same. Every sequence of input symbols accepted by the automaton with the state names A, B, C, D, E will also be accepted by the automaton with the state names $0, 50, 100, 150, 200$ and, conversely, every sequence of input symbols rejected by the automaton with the state names A, B, C, D, E will also be rejected by the automaton with the state names $0, 50, 100, 150, 200$.

Remark 5.1: Are the two automata that just differ in the names of states the same?

We cannot say that the two DFAs, the DFA with the state names A, B, C, D, E and the DFA with the state names $0, 50, 100, 150, 200$, are "the same" or "equal". Recall that a DFA is a 5-tuple, and that two tuples are equal only if all of their components are the same. Since the first components of our two DFAs are the sets of states, and since these two sets are different (one being $\{A, B, C, D, E\}$ and the other being $\{0, 50, 100, 150, 200\}$), the DFAs are not equal. But we can say that they are "equivalent", in a way that exactly the same inputs are accepted by both of them.

Remark 5.2: States Names Convention

We will usually follow a convention where we name the states of an automaton $q_0, q_1, q_2, \ldots$ (or sometimes $p_0, p_1, p_2, \ldots$ or $r_0, r_1, r_2, \ldots$), where q_0 (or p_0 or r_0) will be a starting state.

Alphabet. The set Σ is the alphabet of our automaton, which is the set of all possible input symbols. Here we usually do not have any freedom, as the alphabet of our DFA will either be given to us, or it will be obvious from the problem itself. In our example above, the alphabet is the set of all different kinds of coins that our automaton is expected to deal with. Receiving anything else as an input (such as a £5 note) would not make the automaton merely ignore this input, but would actually *break* the automaton, in that it wouldn't be able to proceed (due to not knowing to what state to move).

Transition Function. Transition function δ is the most interesting part of the definition of a DFA. This is the part that describes how the DFA operates, i.e. how its state changes as it reads the input symbols. The transition function takes as an argument a *pair* of a state p and an input symbol a, and returns a state q. We then say that the DFA *moves* from state p to state q on input symbol a. We also say that there is a *transition* from state p to state q on input symbol a. Note that p and q need not be different; it is possible for a DFA to

stay in the same state on receiving an input. Indeed, this is the case we had in the state 200 of our example automaton, when reading any input would make the automaton remain in that same state.

Another thing to note is that δ is a function with the domain $Q \times \Sigma$. This means that the transition function δ has to be defined for *every* pair (p, a) of state p and input symbol a. That is, for *every* state and *every* input symbol, there has to exist a transition from that state on that input symbol. Even in the situation where we somehow know that a certain input will never be read in a certain state, we still have to define a transition from that state on that input symbol.

The transition function for our example automaton can be given by a Table 5.2

δ	50p	£1	£2
0	50	100	200
50	100	150	200
100	150	200	200
150	200	200	200
200	200	200	200

Table 5.1: Parking Ticket Machine DFA Transition Function

Start State. Or *starting state*, this is simply a state in which our DFA starts processing an input string.

Accepting States. The set of accepting states is the set of states for which, if the DFA ends up in them after reading the whole string, the string is accepted. Accepting states are sometimes called the *final states*, which can be a bit misleading, as the operation of a DFA does not necessarily end when an accepting/final state is entered. Rather, the operation ends when the whole input string has been read. It is perfectly possible (and common) for an accepting state to be entered, and then in the next step to move to a non-accepting state. Therefore, an accepting state by no mean has to be very final. For this reason, we will try to avoid calling accepting states the final states. Note also that the set of accepting states can be empty. In this case, the DFA does not accept any string. These are not particularly interesting DFAs, for obvious reasons, but they are still possible.

The main questions that we will be interested with regard to DFAs is whether a given string is accepted by a given DFA and what are all the strings that are accepted by a given DFA. Most of the times, our task will be to design a DFA (which is to say, define all five of its components) that accepts a given language. Recall that a language is a set of strings, so this comes down to designing a DFA that accepts a given set of string.

Coming back to our example of a parking ticket machine, the DFA for it can formally be defined as $M = (Q, \Sigma, \delta, 0, F)$ where

- $Q = \{0, 50, 100, 150, 200\}$
- $\Sigma = \{50\text{p}, £1, £2\}$
- δ is given by a table

δ	50p	£1	£2
0	50	100	200
50	100	150	200
100	150	200	200
150	200	200	200
200	200	200	200

- Start state is the state 0
- $F = \{200\}$

Video 5.1: Strings Ending With 01

Design a DFA that will accept the strings over alphabet $\{0, 1\}$ that end with substring 01.

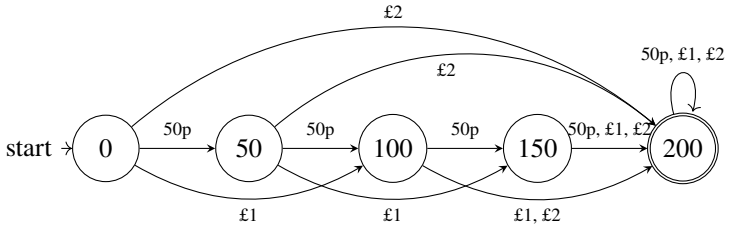
5.3 Representing Deterministic Finite Automata

Representing DFAs by 5-tuples, together with a definition of each of the five components, such as what we did in the example of a parking ticket machine, is quite cumbersome and many times unnecessarily complicated. It is necessary for a formal reasoning about DFAs, but if we just want to visualise how a given DFA works, it is enough to show a transition function in some way and annotate the starting and the accepting states in it. If we look at the Table 5.2 that showed the transition function for our parking ticket machine DFA, we can see that this table really gives us almost all of the information that we need. The labels of the rows of the table are the states, the labels of the columns are the possible input symbols, and the entries themselves are the values of transition function. All we need for a complete description of the DFA is to denote the start state and the set of accepting states. We can do this by adding right arrow ($\rightarrow$) to the start state, and the star ($*$) to each of the accepting states. This is a representation of a DFA by a *transition table*.

For the parking ticket machine DFA, the transition table representation of this DFA is given by a table

δ	50p	£1	£2
$\rightarrow$0	50	100	200
50	100	150	200
100	150	200	200
150	200	200	200
$*$200	200	200	200

Another convenient way of representing a DFA is by a *transition graph*, which we have already demonstrated on the example of the parking ticket machine DFA. In a transition graph, nodes of a graph represent states of the DFA. Directed edges (arrows) represent transitions between states, with the labels of the arrows representing input symbols. The starting state is represented by an arrow labeled "start" going to it. Every final state is represented by a double circle. This is the representation of a DFA that we will most often use. For convenience, here is again the transition graph representation of the parking ticket machine DFA.



5.4 Acceptance of a String by a DFA

Given a formal definition of a DFA, we already talked about a notion of some string being *accepted* by that DFA. We assumed that a string w is accepted by a DFA M if M ends up in an accepting state when it finishes reading w. While this definition of an acceptance of a string by a DFA is pretty intuitive and understandable, it is not precise enough for formal reasoning. To prove that some string is accepted by a given DFA, we would need to calculate the exact state in which the DFA ends up being after the whole string is read, which is particularly laboursome if the string is very long. And answering a question like “what are all the strings that are accepted by this particular DFA” is next to impossible.

To make our life easier, we need some notion of a *step* of processing a string by a DFA. Fortunately, we already have something that formally defines a computational step of a DFA - the transition function! Thus, if $\delta(p, a) = q$, we say that the automaton moves from state p to state q when it reads (or processes) the symbol a *in one step*. Now we need some similar definition of a state that the automaton moves to when it reads any part of the string (including the complete string).

Definition 5.2: Transition Function for Strings - Informal

For the DFA $M = (Q, \Sigma, \delta, q_0, F)$, transition function $\hat{\delta}$ for strings over Σ returns, for a given state q and string w, the state in which the automaton M is when it reads the whole string w, if it starts reading in the state q.

Definition 5.3: Transition Function for Strings - Formal

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We define the *transition function for strings over* Σ, $\hat{\delta} : Q \times \Sigma^* \to Q$ in the following way:

1. If $w = \epsilon$, then $\hat{\delta}(q, w) = q$, for every state $q \in Q$.
2. If $w = xa$, where $x \in \Sigma^*$ and $a \in \Sigma$, then $\hat{\delta}(q, w) = \hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$, for every state $q \in Q$.

Let us analyse this definition now. Firstly, note that we denote the transition function for strings by $\hat{\delta}$ and that the domain of this function is $Q \times \Sigma^*$ (rather than $Q \times \Sigma$, as for the transition function on symbols, which is a part of the definition of a DFA). This means that this function takes a state and a string over alphabet Σ as an input. Similarly to the transition function on symbols, it returns a state as an output. We can also see the similarity between this definition and the principle of mathematical induction that we looked at in Section 3.4.3. As in mathematical induction, here we also have a *base case* and a *step*. This is the first of many *inductive definitions* that we will encounter in our course. The "induction" in our definition is over the length of a string in question.

The *base case* of our inductive definition is the case of a string with minimal length - that is, of length 0. We remember that there is only one string of length 0 - namely, the empty string ϵ. This is the first part of the definition, which simply says that if the string has length 0, then, after processing the string, the DFA stays in the same state it was previously in ($\hat{\delta}(q, w) = q$, when $w = \epsilon$). This is not surprising, as this is essentially the case of our automaton not receiving any input (and, therefore, not moving anywhere from its current state).

The second part of the definition is equivalent to the induction step. It essentially says the following: if we know the next-to-last state of the DFA when processing the string w (that is, the state in which the DFA ends up after it processes all but the last symbol of the string w), then the state in which it will end up after processing the whole string w is determined simply by the transition from the next-to-last state on the last symbol of string w. This is very intuitive - to determine in which state the DFA ends up after it processes the whole string w, we first determine in which state it ends after it processes the string w without the last symbol, and then we make one more transition from that state on the last symbol. This is captured formally by the second part of the definition - if the string w has length more than 1, then it can be written as xa, where a is the last symbol of string and x is the prefix of the string. Note that x can also be ϵ. Then, the state in which the DFA is after processing all but last symbol of w, if it starts in the state q, is $\hat{\delta}(q, x)$. Let us call this state p. The state in which the DFA will be after it processes the whole string w is then the state to which the DFA moves from p on the last symbol of w, which is a. This is $\delta(p, a)$ (or, if we remember what p is, $\delta(\hat{\delta}(q, x), a)$. Notice that in this part we have both the transition function for a symbol (δ) and the transition function for a string $\hat{\delta}$.

Note the principle of mathematical induction here. To calculate the transition function over a string of length n for some starting state q, we reduce the problem to a smaller problem of calculating the transition function over a string of length $n - 1$, which we assume we know how to do. Note also that this definition is given in terms of reducing the problem to the problem of a smaller size until we reach the base case of a string of length 0. This is, in a way, going *backward* from a full string to an empty string. The way we intuitively think about computing the state in which a DFA is when it reads the complete string is exactly the opposite of this. We think of starting from the start state, calculating where the DFA is after it reads the first symbol of the string, then where it goes from there after reading the second symbol and so on. In essence, this is going *forward* in a string.

Video 5.2: Strings Ending With 01

For a DFA in Video 5.1, calculate the $\hat{\delta}(00010)$ and $\hat{\delta}(000101)$.

One thing to note is that the Definition 5.3 also covers the strings consisting of a single symbol; that is, the strings $w = a$, where a is a symbol of the alphabet Σ. So, $\hat{\delta}(q, a)$ is the transition function for a state q and a *string* that consists of a single symbol a. This is subtly different from a transition function $\delta(q, a)$ which is the transition function for a state q and a *symbol* a[1]. It happens to be the case that these two functions always return the same state for DFAs. That is, $\hat{\delta}(q, a) = \delta(q, a)$, for every state q and every symbol a. We

[1]The string a and the symbol a are two different things in the same way that the sequence of one symbol (a) is different from just a symbol a.

could, therefore, use the symbol δ for both of these functions, because no confusion can ever arise about their use (in turn, because $\hat{\delta}$ and δ return the same result for the "same" input[2]). This is, in fact, done in most of the literature. However, we still think it is useful to keep the distinction between the two transition functions in mind. Secondly, while they return the same value for the "same" input for DFAs, this will not be the case with some other kinds of automata, and there we would have to use different symbols, possibly causing confusion. For these reasons, we are going to keep denoting them using different symbols, $\hat{\delta}$ and δ.

5.5 Language Accepted by a DFA and Regular Languages

Now that we have introduced a definition of an acceptance of a string by a DFA, we can finally define formally the language accepted by a DFA. Informally, the language accepted by a DFA is the set of all string accepted by that DFA. Using the transition function for strings, we can define this more formally.

Definition 5.4: Language Accepted by a DFA

For the DFA $M = (Q, \Sigma, \delta, q_0, F)$, the language accepted by M (usually denoted by $L(M)$ or L_M) is the set of all strings w over Σ, such that $\hat{\delta}(q_0, w) \in F$

$$L(M) = \{w | \hat{\delta}(q_0, w) \in F\}.$$

Note that if the language L is accepted by a DFA M, it means that M accepts *all* the strings from L and *only* the strings in L. Taking our Parking Ticket Machine DFA as an example, we cannot say that, for example, $L = \{fpf, fffp\}$ is the language accepted by that DFA because, although all the strings in that language are accepted by the DFA, there are many strings that are also accepted by the DFA that are not in L.

Definition 5.5: Regular Language

If the language L is accepted by some DFA M, then we say that language L is *regular*.

Remark 5.3

Regularity is the property of a *language*, rather than of an *automaton*. That is, the language is regular if there exist any DFA that accepts it.

[2]The "same" input is in the quotation here because the inputs aren't really the same. See the previous footnote.