Automata and Computability

Solutions to Exercises

Fall 2016

Alexis Maciel
Department of Computer Science
Clarkson University

Copyright © 2016 Alexis Maciel

Contents

Preface

This document contains solutions to the exercises of the course notes *Automata and Computability*. These notes were written for the course CS345 *Automata Theory and Formal Languages* taught at Clarkson University. The course is also listed as MA345 and CS541. The solutions are organized according to the same chapters and sections as the notes.

Here's some advice. Whether you are studying these notes as a student in a course or in self-directed study, your goal should be to understand the material well enough that you can do the exercises on your own. Simply studying the solutions is not the best way to achieve this. It is much better to spend a reasonable amount of time and effort trying to do the exercises yourself before looking at the solutions.

If you can't do an exercise on your own, you should study the notes some more. If that doesn't work, seek help from another student or from your instructor. Look at the solutions only to check your answer once you think you know how to do an exercise.

If you needed help doing an exercise, try redoing the same exercise later on your own. And do additional exercises.

If your solution to an exercise is different from the official solution, take the time to figure out why. Did you make a mistake? Did you forget something?

Did you discover another correct solution? If you're not sure, ask for help from another student or the instructor. If your solution turns out to be incorrect, fix it, after maybe getting some help, then try redoing the same exercise later on your own and do additional exercises.

Feedback on the notes and solutions is welcome. Please send comments to `alexis@clarkson.edu`.

Chapter 1

Introduction

There are no exercises in this chapter.

Chapter 2

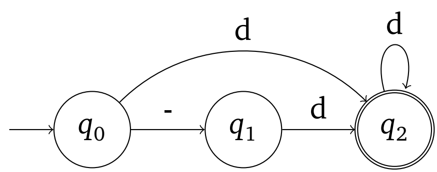
Finite Automata

2.1 Turing Machines

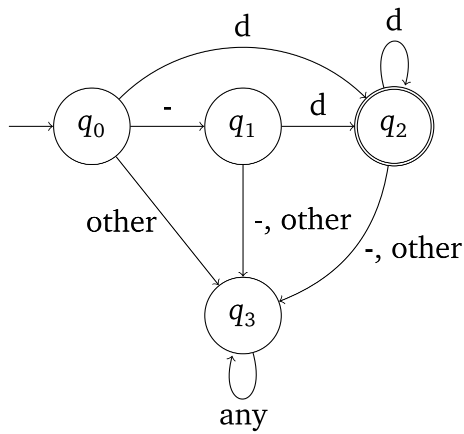
There are no exercises in this section.

2.2 Introduction to Finite Automata

2.2.3.

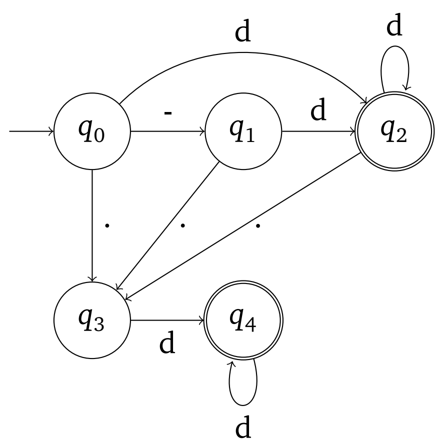


Missing edges go to a garbage state. In other words, the full DFA looks like this:



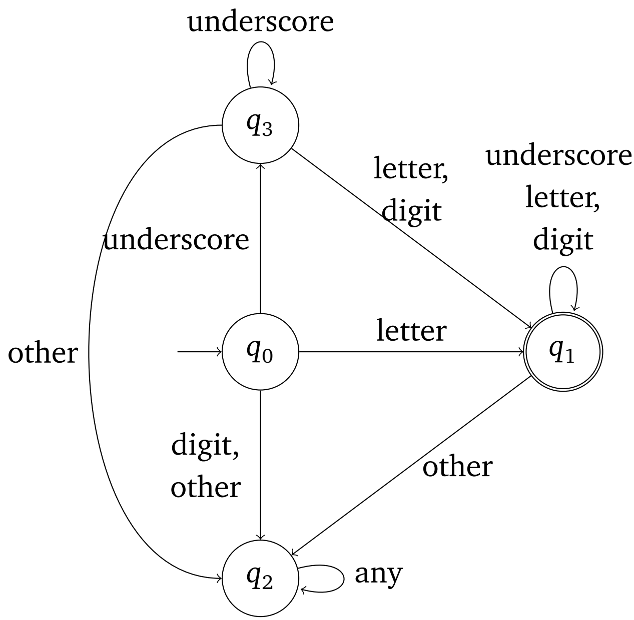
The transition label *other* means any character that's not a dash or a digit.

2.2.4.

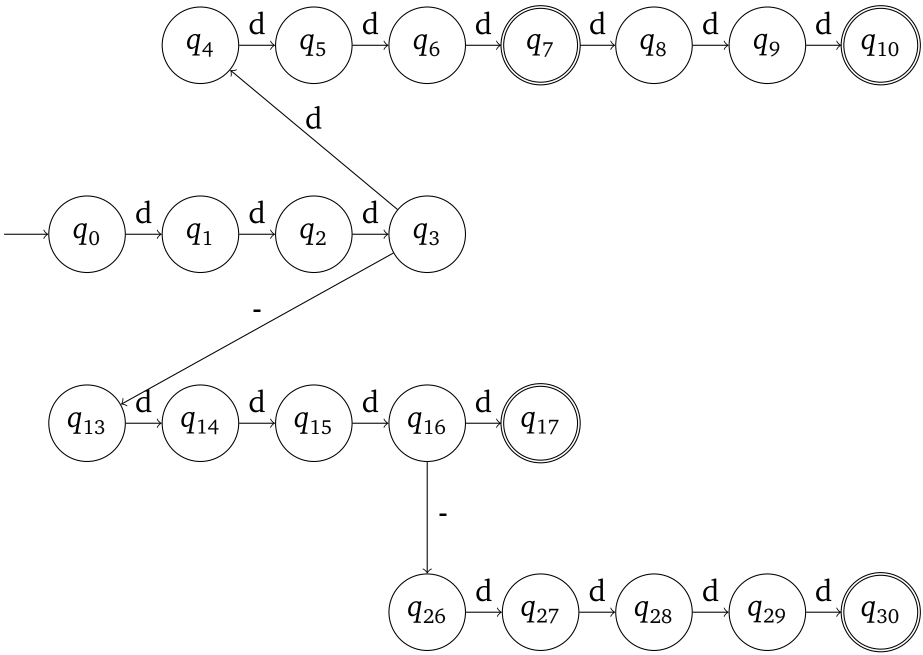


Missing edges go to a garbage state.

2.2.5.


underscore
q_3
letter,
digit
underscore,
letter,
digit
underscore
other
q_0
letter
q_1
digit,
other
other
q_2
any

2.2.6.


d
d
d
d
d
d
d
d
d
d
-
d
d
d
d
-
d
d
d
d
q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
q_8
q_9
q_{10}
q_{13}
q_{14}
q_{15}
q_{16}
q_{17}
q_{26}
q_{27}
q_{28}
q_{29}
q_{30}

2.2.7.

```
starting_state() { return q0 }

is_accepting(q) { return true iff q is q1 }

next_state(q, c) {
    if (q is q0)
        if (c is underscore or letter)
            return q1
        else
            return q2
    else if (q is q1)
        if (c is underscore, letter or digit)
            return q1
        else
            return q2
    else // q is q2
        return q2
}
```

2.2.8. The following assumes that the garbage state is labeled q_9. In the pseudocode algorithm, states are stored as integers. This is more convenient here.

```
starting_state() { return 0 }

is_accepting(q) { return true iff q is 8 }
```

```
next_state(q, c) {
    if (q in {0, 1, 2} or {4, 5, 6, 7})
        if (c is digit)
            return q + 1
        else
            return 9
    else if (q is 3)
        if (c is digit)
            return 5
        else if (c is dash)
            return 4
        else
            return 9
    else if (q is 8 or 9)
        return 9
}
```

2.3 Formal Definition

2.3.9. The DFA is $(\{q_0, q_1, q_2, \ldots, q_9\}, \Sigma, \delta, q_0, \{q_8\})$ where Σ is the set of all characters that appear on a standard keyboard and δ is defined as follows:

$$\delta(q_i, c) = \begin{cases} q_{i+1} & \text{if } i \notin \{3,8,9\} \text{ and } c \text{ is digit} \\ q_9 & \text{if } i \notin \{3,8,9\} \text{ and } c \text{ is not digit} \end{cases}$$

$$\delta(q_3, c) = \begin{cases} q_4 & \text{if } c \text{ is dash} \\ q_5 & \text{if } c \text{ is digit} \\ q_9 & \text{otherwise} \end{cases}$$

$$\delta(q_8, c) = q_9 \quad \text{for every } c$$

$$\delta(q_9, c) = q_9 \quad \text{for every } c$$

2.3.10. The DFA is $(\{q_0, q_1, q_2, q_3\}, \Sigma, \delta, q_0, \{q_2\})$ where Σ is the set of all characters that appear on a standard keyboard and δ is defined as follows:

$$\delta(q_0, c) = \begin{cases} q_1 & \text{if } c \text{ is dash} \\ q_2 & \text{if } c \text{ is digit} \\ q_3 & \text{otherwise} \end{cases}$$

$$\delta(q_i, c) = \begin{cases} q_2 & \text{if } i \in \{1,2\} \text{ and } c \text{ is digit} \\ q_3 & \text{if } i \in \{1,2\} \text{ and } c \text{ is not digit} \end{cases}$$

$$\delta(q_3, c) = q_3 \quad \text{for every } c$$

2.3.11. The DFA is $(\{q_0, q_1, q_2, \ldots, q_5\}, \Sigma, \delta, q_0, \{q_2, q_4\})$ where Σ is the set of all characters that appear on a standard keyboard and δ is defined as follows:

$$\delta(q_0, c) = \begin{cases} q_1 & \text{if } c \text{ is dash} \\ q_2 & \text{if } c \text{ is digit} \\ q_3 & \text{if } c \text{ is decimal point} \\ q_5 & \text{otherwise} \end{cases}$$

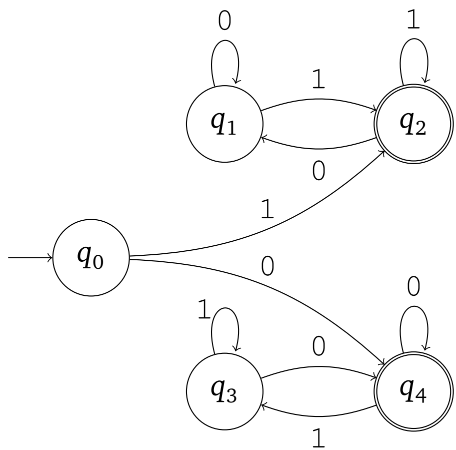
$$\delta(q_i, c) = \begin{cases} q_2 & \text{if } i \in \{1, 2\} \text{ and } c \text{ is digit} \\ q_3 & \text{if } i \in \{1, 2\} \text{ and } c \text{ is decimal point} \\ q_5 & \text{if } i \in \{1, 2\} \text{ and } c \text{ is not digit or decimal point} \end{cases}$$

$$\delta(q_i, c) = \begin{cases} q_4 & \text{if } i \in \{3, 4\} \text{ and } c \text{ is digit} \\ q_5 & \text{if } i \in \{3, 4\} \text{ and } c \text{ is not digit} \end{cases}$$

$$\delta(q_5, c) = q_5 \quad \text{for every } c$$

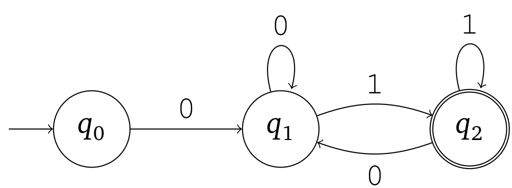
2.4 More Examples

2.4.1.

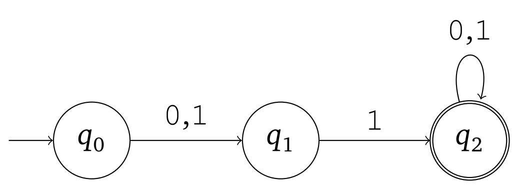


2.4.2. In all cases, missing edges go to a garbage state.

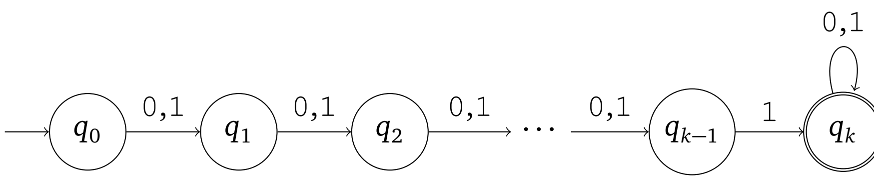
a)



b)

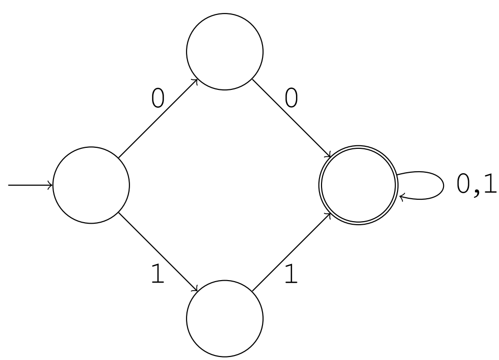


c)

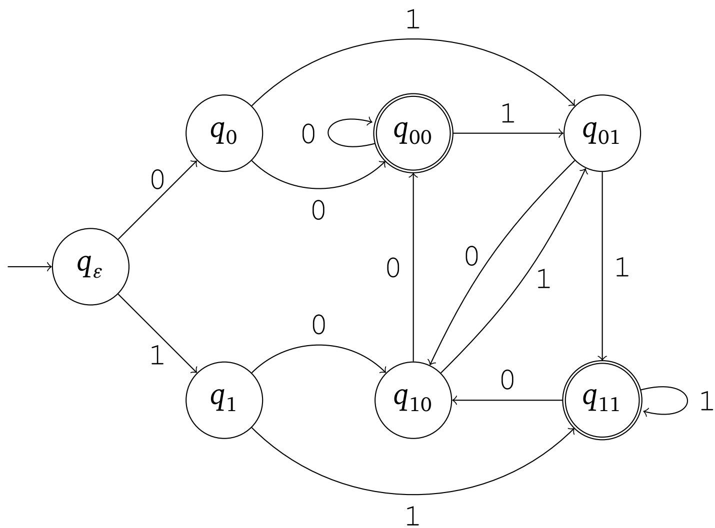


2.4.3. In all cases, missing edges go to a garbage state.

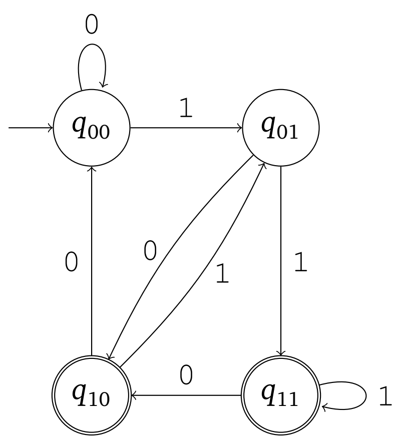
a)



b) The idea is for the DFA to remember the last two symbols it has seen.



c) Again, the idea is for the DFA to remember the last two symbols it has seen. We could simply change the accepting states of the previous DFA to $\{q_{10}, q_{11}\}$. But we can also simplify this DFA by assuming that strings of length less than two are preceded by `00`.



d) The idea is for the DFA to remember the last k symbols it has seen. But this is too difficult to draw clearly, so here's a formal description of the DFA: $(Q, \{\texttt{0}, \texttt{1}\}, \delta, q_0, F)$ where

$$Q = \{q_w \mid w \in \{\texttt{0}, \texttt{1}\}^* \text{ and } w \text{ has length } k\}$$
$$q_0 = q_{w_0} \quad \text{where } w_0 = \texttt{0}^k \text{ (that is, a string of } k \text{ } \texttt{0}\text{'s)}$$
$$F = \{q_w \in Q \mid w \text{ starts with a } \texttt{1}\}$$

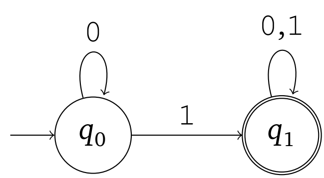
and δ is defined as follows:

$$\delta(q_{au}, b) = q_{ub}$$

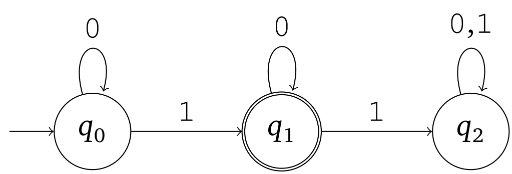
where $a \in \Sigma$, u is a string of length $k-1$ and $b \in \Sigma$.

2.4.4. In all cases, missing edges go to a garbage state.

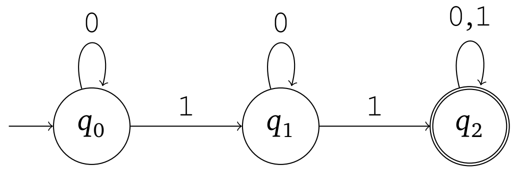
a)



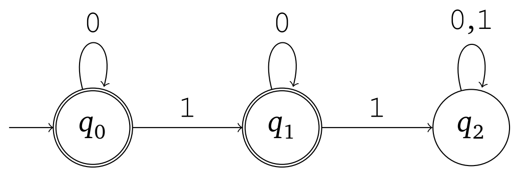
b)



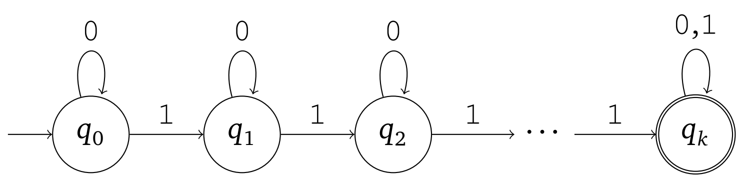
c)



d)

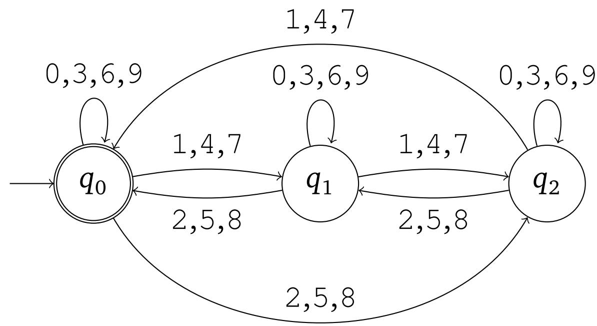


e)



2.4.5.

a) The idea is for the DFA to store the value, modulo 3, of the portion of the number it has seen so far, and then update that value for every additional digit that is read. To update the value, the current value is multiplied by 10, the new digit is added and the result is reduced modulo 3.



(Note that this is exactly the same DFA we designed in an example of this section for the language of strings that have the property that the sum of their digits is a multiple of 3. This is because 10 mod 3 = 1 so that when we multiply the current value by 10 and reduce modulo 3, we are really just multiplying by 1. Which implies that the strategy we described above is equivalent to simply adding the digits of the number, modulo 3.)

b) We use the same strategy that was described in the first part, but this time, we reduce modulo k. Here's a formal description of the DFA: $(Q, \Sigma, \delta, q_0, F)$ where

$$Q = \{q_0, q_1, q_2, \ldots, q_{k-1}\}$$
$$\Sigma = \{\texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9}\}$$
$$F = \{q_0\}$$

and δ is defined as follows: for every $i \in Q$ and $c \in \Sigma$,

$$\delta(q_i, c) = q_j \quad \text{where } j = (i \cdot 10 + c) \bmod k.$$

2.4.6.

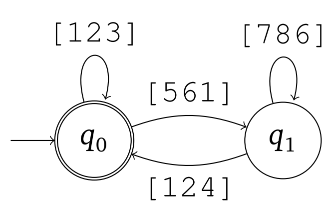
a) The idea is for the DFA to verify, for each input symbol, that the third digit is the sum of the first two plus any carry that was previously generated, as well as determine if a carry is generated. All that the DFA needs to remember is the value of the carry (0 or 1). The DFA accepts if no carry is generated when processing the last input symbol. Here's a formal description of the DFA, where state q_2 is a garbage state: $(Q, \Sigma, \delta, q_0, F)$ where

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2\} \\
\Sigma &= \{\,\texttt{[}abc\texttt{]} \mid a, b, c \in \{\texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9}\}\} \\
F &= \{q_0\}
\end{aligned}
$$

and δ is defined as follows:

$$
\delta(q_d, \texttt{[}abc\texttt{]}) = \begin{cases} q_0 & \text{if } d \in \{0,1\} \text{ and } d + a + b = c \\ q_1 & \text{if } d \in \{0,1\},\ d + a + b \geq 10 \text{ and} \\ & \quad (d + a + b) \bmod 10 = c \\ q_2 & \text{otherwise} \end{cases}
$$

Here's a transition diagram of the DFA that shows only one of the 1,000 transitions that come out of each state.



b) Since the DFA is now reading the numbers from left to right, it can't compute the carries as it reads the numbers. So it will do the opposite: for each input symbol, the DFA will figure out what carry it needs from the rest of the numbers. For example, if the first symbol that the DFA sees is [123], the DFA will know that there should be no carry generated from the rest of the numbers. But if the symbol is [124], the DFA needs the rest of the number to generate a carry. And if a carry needs to be generated, the next symbol will have to be something like [561] but not [358]. The states of the DFA will be used to remember the carry that is needed from the rest of the numbers. The DFA will accept if no carry is needed for the first position of the numbers (which is given by the last symbol of the input string). Here's a formal description of the DFA, where state q_2 is a garbage state: $(Q, \Sigma, \delta, q_0, F)$ where

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2\} \\
\Sigma &= \{\,\texttt{[}abc\texttt{]} \mid a, b, c \in \{\texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9}\}\} \\
F &= \{q_0\}
\end{aligned}
$$

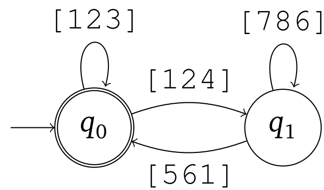
and δ is defined as follows:

$$
\delta(q_0, \texttt{[}abc\texttt{]}) = \begin{cases} q_d & \text{if } d \in \{0, 1\} \text{ and } d + a + b = c \\ q_2 & \text{otherwise} \end{cases}
$$

$$
\delta(q_1, \texttt{[}abc\texttt{]}) = \begin{cases} q_d & \text{if } d \in \{0, 1\},\ d + a + b \geq 10 \text{ and} \\ & \quad (d + a + b) \bmod 10 = c \\ q_2 & \text{otherwise} \end{cases}
$$

$$
\delta(q_2, \texttt{[}abc\texttt{]}) = q_2, \quad \text{for every } \texttt{[}abc\texttt{]} \in \Sigma
$$

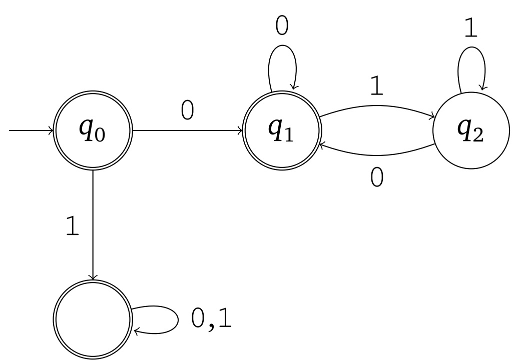
Here's a transition diagram of the DFA that shows only one of the 1,000 transitions that come out of each state.



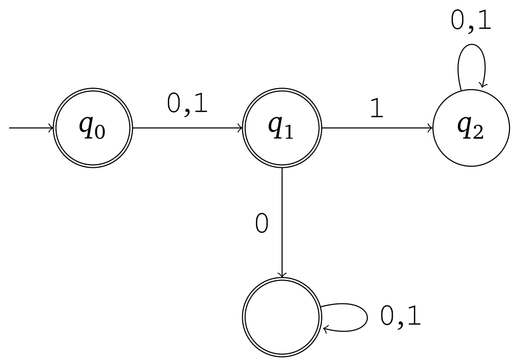
2.5 Closure Properties

2.5.3. In each case, all we have to do is switch the acceptance status of each state. But we need to remember to do it for the garbage states too.

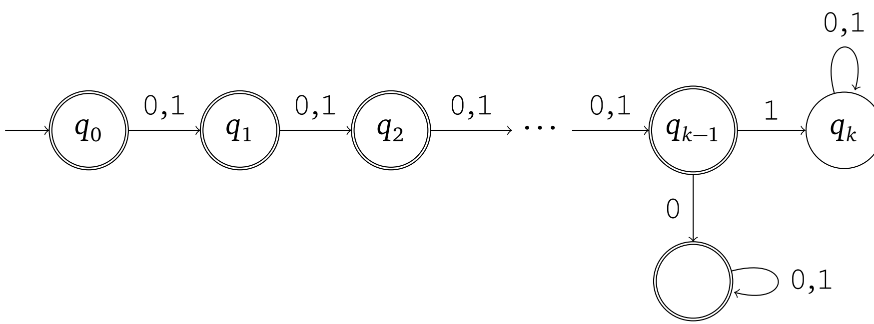
a)



b)

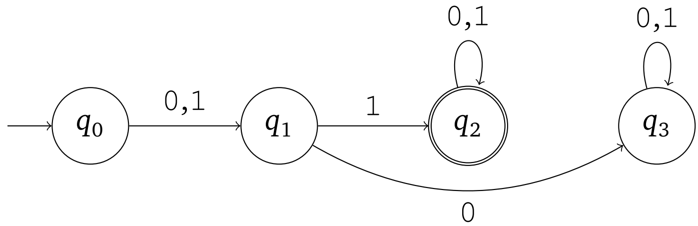


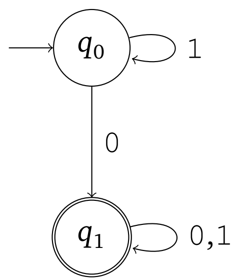
c)

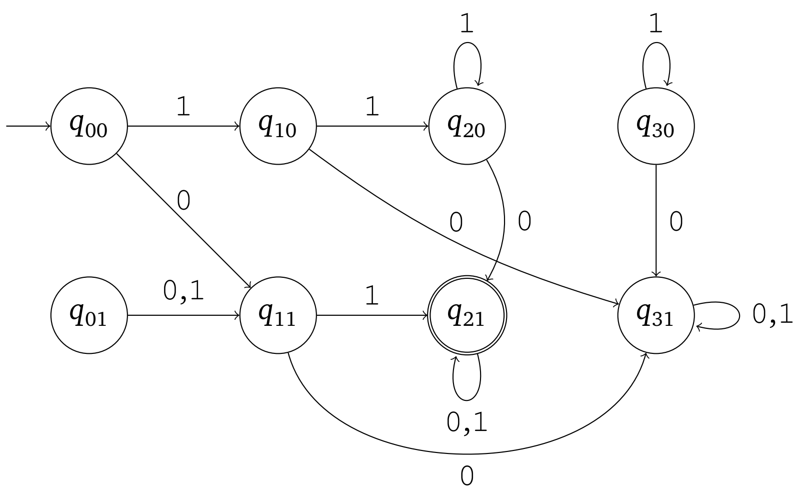


2.5.4. It is important to include in the pair construction the garbage states of the DFA's for the simpler languages. (This is actually not needed for intersections but it is critical for unions.) In each case, we give the DFA's for the two simpler languages followed by the DFA obtained by the pair construction.

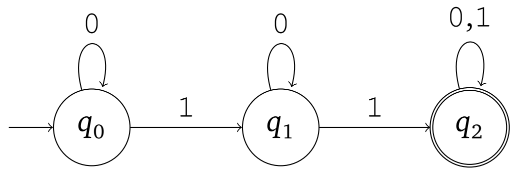
a)

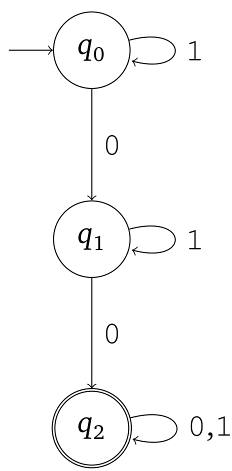

q_0
0,1
q_1
1
q_2
0,1
q_3
0,1
0

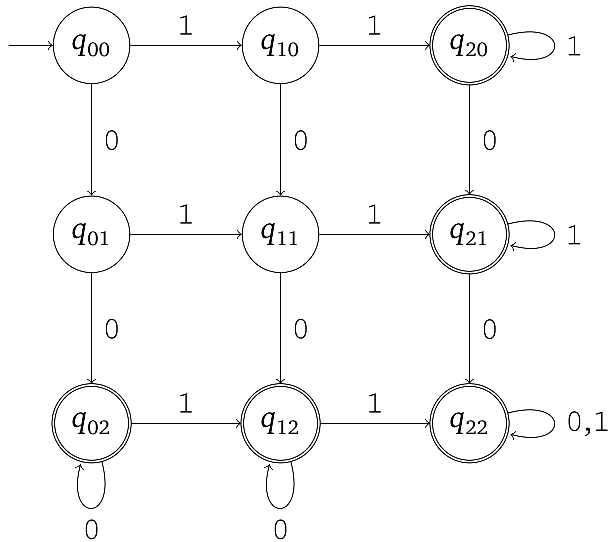

q_0
1
0
q_1
0,1


q_{00}
1
q_{10}
1
q_{20}
1
q_{30}
1
0
0
0
0
q_{01}
0,1
q_{11}
1
q_{21}
q_{31}
0,1
0,1
0

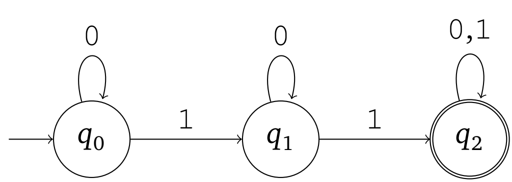
b)

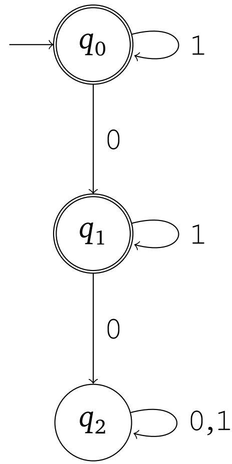


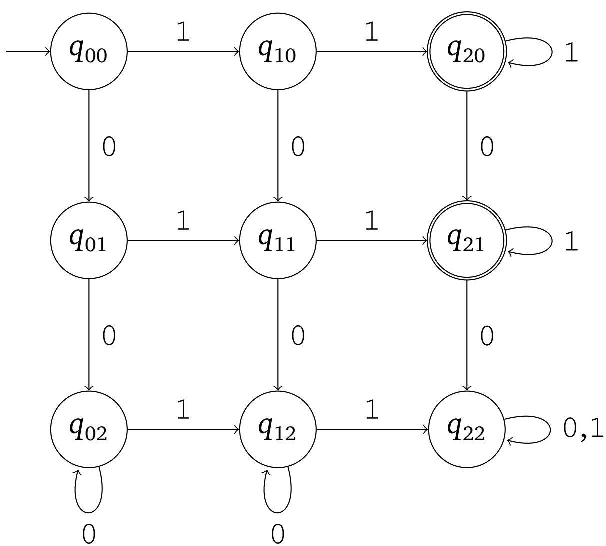




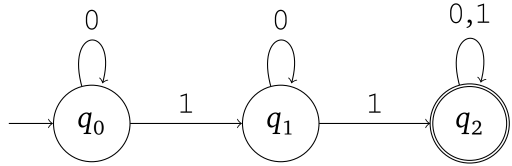
c)

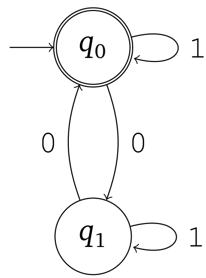


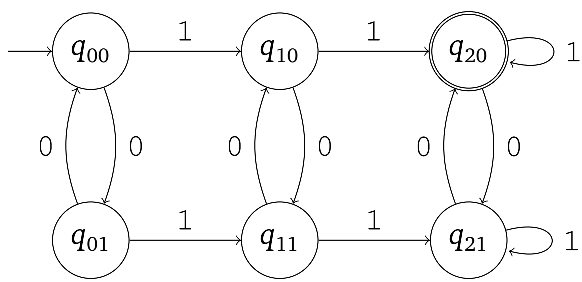

q_0
1
0
q_1
1
0
q_2
0,1


q_{00}
1
q_{10}
1
q_{20}
1
0
0
0
q_{01}
1
q_{11}
1
q_{21}
1
0
0
0
q_{02}
1
q_{12}
1
q_{22}
0,1
0
0

d)

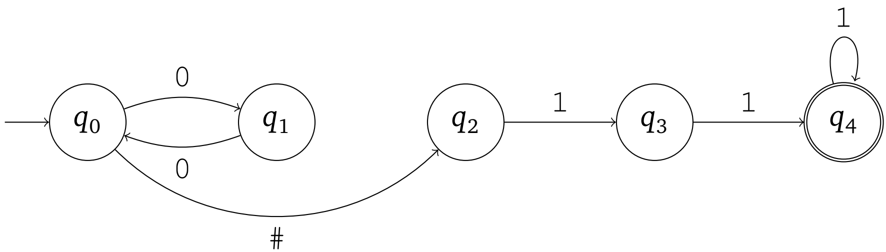

0
0
0,1
1
1
q_0
q_1
q_2


q_0
1
0
0
q_1
1

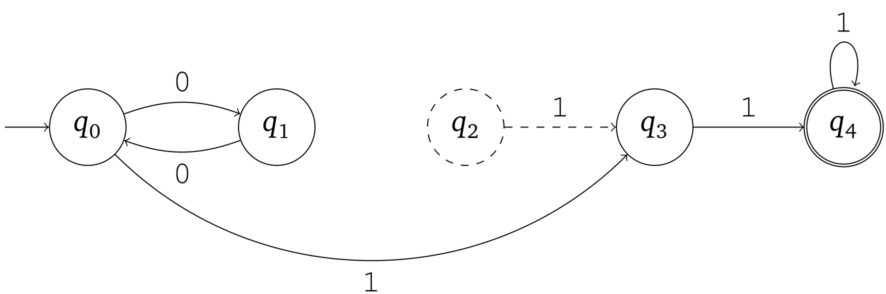

q_{00}
1
q_{10}
1
q_{20}
1
0
0
0
0
0
0
q_{01}
1
q_{11}
1
q_{21}
1

2.5.5. In both cases, missing edges go to a garbage state.

a)



b) The dashed state and edge could be deleted.



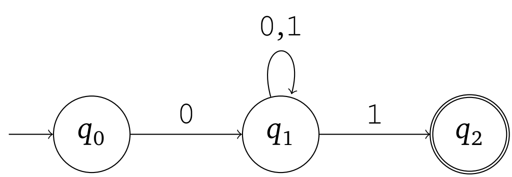
Chapter 3

Nondeterministic Finite Automata

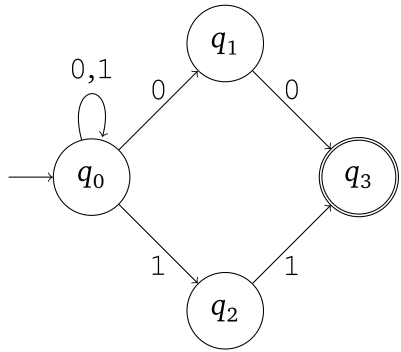
3.1 Introduction

3.1.3.

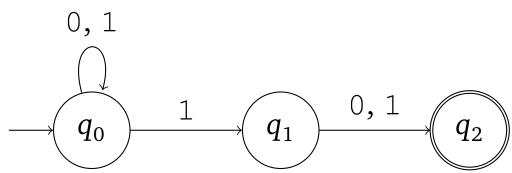
a)



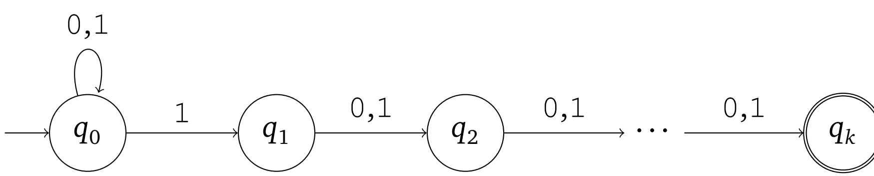
b)



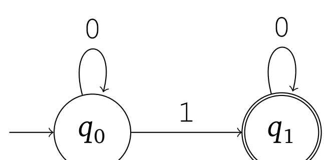
c)



d)



e)



3.2 Formal Definition

3.2.1. The NFA is $(Q, \{\mathtt{0}, \mathtt{1}\}, \delta, q_0, F)$ where

$$Q = \{q_0, q_1, q_2, q_3\}$$
$$F = \{q_3\}$$

and δ is defined by the following table

δ	$\mathtt{0}$	$\mathtt{1}$	ε
q_0	q_0	q_0, q_1	—
q_1	q_2	q_2	—
q_2	q_3	q_3	—
q_3	—	—	—

3.2.2. The NFA is $(Q, \{\mathtt{0}, \mathtt{1}\}, \delta, q_0, F)$ where

$$Q = \{q_0, q_1, q_2, q_3\}$$
$$F = \{q_3\}$$

and δ is defined by the following table:

δ	0	1	ε
q_0	q_1	q_0	$-$
q_1	q_2	$-$	q_0
q_2	$-$	q_3	q_1
q_3	q_3	q_3	$-$

3.2.3.

$$
\begin{array}{ccccccccccccc}
q_0 & \xrightarrow{0} & q_0 & \xrightarrow{0} & q_0 & \xrightarrow{1} & q_0 & \xrightarrow{0} & q_0 & \xrightarrow{0} & q_0 & \xrightarrow{1} & q_0 \\
q_0 & & q_0 & & q_0 & & q_0 & & q_1 & & q_2 & & q_3 \\
q_0 & & q_1 & & q_2 & & q_3 & & q_3 & & q_3 & & q_3
\end{array}
$$

The NFA accepts because the last two sequences end in the accepting state.

3.2.4.

$$q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{1} q_0$$

$$q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{1} q_0$$

$$q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{1} q_0$$

$$q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{1} q_0$$

$$q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{1} q_3$$

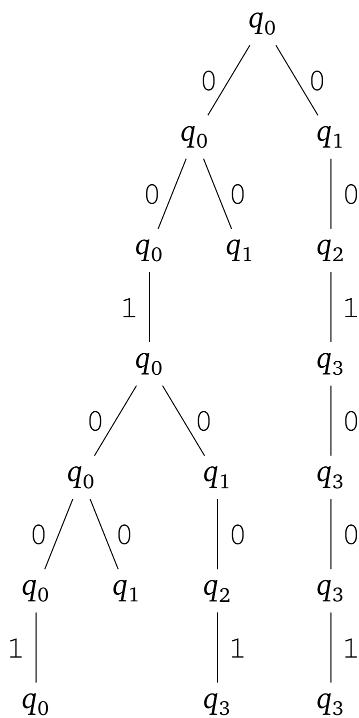
$$q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{1} q_3$$

$$q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{1} q_3 \xrightarrow{0} q_3 \xrightarrow{0} q_3 \xrightarrow{1} q_3$$

The NFA accepts because the last three sequences end in the accepting state.

3.3 Equivalence with DFA's

3.3.2.



3.3.3.

a)

δ'	0	1
q_0	q_1	$-$
q_1	q_1	q_1, q_2
q_2	$-$	$-$
q_1, q_2	q_1	q_1, q_2

The start state is $\{0\}$. The accepting state is $\{q_1, q_2\}$. (State $\{q_2\}$ is unreachable from the start state.) Missing transitions go to the garbage state ($-$).

b)

δ'	0	1
q_0	q_0, q_1	q_0, q_2
q_1	q_3	$-$
q_2	$-$	q_3
q_3	$-$	$-$
q_0, q_1	q_0, q_1, q_3	q_0, q_2
q_0, q_2	q_0, q_1	q_0, q_2, q_3
q_0, q_1, q_3	q_0, q_1, q_3	q_0, q_2
q_0, q_2, q_3	q_0, q_1	q_0, q_2, q_3

The start state is $\{q_0\}$. The accepting states are $\{q_0, q_1, q_3\}$ and $\{q_0, q_2, q_3\}$. (States $\{q_1\}$, $\{q_2\}$ and $\{q_3\}$ are unreachable from the start state.)

c)

δ'	0	1
q_0	q_0	q_0, q_1
q_1	q_2	q_2
q_2	—	—
q_0, q_1	q_0, q_2	q_0, q_1, q_2
q_0, q_2	q_0	q_0, q_1
q_0, q_1, q_2	q_0, q_2	q_0, q_1, q_2

The start state is $\{q_0\}$. The accepting states are $\{q_0, q_2\}$ and $\{q_0, q_1, q_2\}$. (States $\{q_1\}$ and $\{q_2\}$ are unreachable from the start state.)

d)

δ'	0	1
q_0	q_0	q_1
q_1	q_1	—

The start state is $\{q_0\}$. The accepting state is $\{q_1\}$. Missing transitions go to the garbage state (—). (The given NFA was almost a DFA. All that was missing was a garbage state and that's precisely what the algorithm added.)

3.3.4.

a)

δ'	0	1
q_0	q_1	$-$
q_1	q_1	q_1, q_2
q_2	$-$	$-$
q_1, q_2	q_1	q_1, q_2

The start state is $E(\{q_0\}) = \{q_0\}$. The accepting state is $\{q_1, q_2\}$. (State $\{q_2\}$ is unreachable from the start state.) Missing transitions go to the garbage state ($-$).

b)

δ'	0	1
q_0	q_0, q_1, q_2	q_0, q_1, q_2
q_1	q_3	$-$
q_2	$-$	q_3
q_3	$-$	$-$
q_0, q_1, q_2	q_0, q_1, q_2, q_3	q_0, q_1, q_2, q_3
q_0, q_1, q_2, q_3	q_0, q_1, q_2, q_3	q_0, q_1, q_2, q_3

The start state is $E(\{q_0\}) = \{q_0, q_1, q_2\}$. The accepting state is $\{q_0, q_1, q_2, q_3\}$. (States $\{q_0\}$, $\{q_1\}$, $\{q_2\}$ and $\{q_3\}$ are unreachable from the start state.)

3.4 Closure Properties

3.4.2. Suppose that $M_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$, for $i = 1, 2$. Without loss of generality, assume that Q_1 and Q_2 are disjoint. Then $N = (Q, \Sigma, \delta, q_0, F)$ where

$$\begin{aligned} Q &= Q_1 \cup Q_2 \\ q_0 &= q_1 \\ F &= F_2 \end{aligned}$$

and δ is defined as follows:

$$\delta(q, \varepsilon) = \begin{cases} \{q_2\} & \text{if } q \in F_1 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\delta(q, a) = \{\delta_i(q, a)\}, \quad \text{if } q \in Q_i \text{ and } a \in \Sigma.$$

3.4.3. Suppose that $M = (Q_1, \Sigma, \delta_1, q_1, F_1)$. Let q_0 be a state not in Q_1. Then $N = (Q, \Sigma, \delta, q_0, F)$ where

$$\begin{aligned} Q &= Q_1 \cup \{q_0\} \\ F &= F_1 \cup \{q_0\} \end{aligned}$$

and δ is defined as follows:

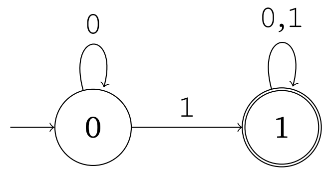
$$\delta(q, \varepsilon) = \begin{cases} \{q_1\} & \text{if } q \in F_1 \cup \{q_0\} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\delta(q, a) = \{\delta_1(q, a)\}, \quad \text{if } q \neq q_0 \text{ and } a \in \Sigma.$$

3.4.4.

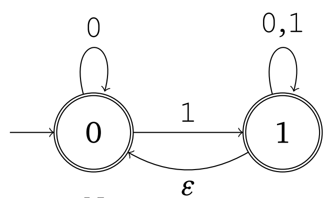
a) In the second to last paragraph of the proof, if $k = 0$, then it is claimed that $w = x_1$ with $x_1 \in A$. We know x_1 is accepted by N, but there is no reason why that can't be because x_1 leads back to the start state instead of leading to one of the original accepting states of M.

b) Consider the following DFA for the language of strings that contain at least one `1`:



M

If we used this idea, we would get the following NFA:



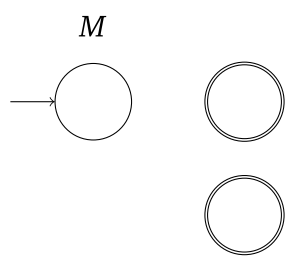
N

This NFA accepts strings that contain only `0`'s. These strings are not in the language $L(M)^*$. Therefore, $L(N) \neq L(M)^*$.

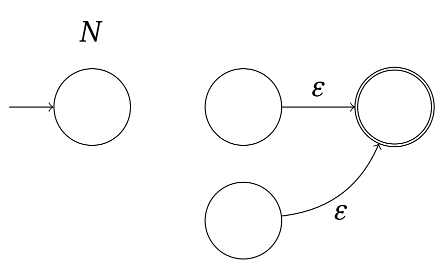
3.4.5. This can be shown by modifying the construction that was used for the star operation. The only change is that a new start state should not be added. The argument that this construction works is almost the same as before. If $w \in A^+$, then $w = x_1 \cdots x_k$ with $k \geq 1$ and each $x_i \in A$. This implies that N can accept w by going through M k times, each time reading one x_i and then returning to the start state of M by using one of the new ε transitions (except after x_k).

Conversely, if w is accepted by N, then it must be that N uses the new ε "looping back" transitions k times, for some number $k \geq 0$, breaking w up into $x_1 \cdots x_{k+1}$, with each $x_i \in A$. This implies that $w \in A^+$. Therefore, $L(N) = A^+$.

3.4.6. Suppose that L is regular and that it is recognized by a DFA M that looks like this:



This DFA can be turned into an equivalent NFA N with a single accepting state as follows:



That is, we add a new accepting state, an ε transition from each of the old accepting states to the new one, and we make the old accepting states non-accepting.

We can show that $L(N) = L(M)$ as follows. If w is accepted by M, then w leads to an old accepting, which implies that N can accept w by using one of the new transitions. If w is accepted by N, then the reading of w must finish with one of the new transitions. This implies that in M, w leads to one of the old accepting states, so w is accepted by M.

3.4.7. Suppose that L is recognized by a DFA M. Transform N into an equivalent NFA with a single accepting state. (The previous exercise says that this can be done.) Now reverse every transition in N: if a transition labeled a goes from q_1 to q_2, make it go from q_2 to q_1. In addition, make the accepting state become the start state, and switch the accepting status of the new and old start states. Call the result N'.

We claim that N' recognizes $L^{\mathscr{R}}$. If $w = w_1 \cdots w_n$ is accepted by N', it must be that there is a path through N' labeled w. But then, this means that there was a path labeled $w_n \cdots w_1$ through N. Therefore, w is the reverse of a string in L, which means that $w \in L^{\mathscr{R}}$. It is easy to see that the reverse is also true.

Chapter 4

Regular Expressions

4.1 Introduction

4.1.5.

a) $(-\cup\varepsilon)DD^*$.

b) $(-\cup\varepsilon)DD^*\cup(-\cup\varepsilon)D^*\,\mathbf{.}\,DD^*$.

c) $\_(\_\cup L\cup D)^*(L\cup D)(\_\cup L\cup D)^*\cup L(\_\cup L\cup D)^*$.

d) $D^7\cup D^{10}\cup D^3{-}D^4\cup D^3{-}D^3{-}D^4$.

4.2 Formal Definition

There are no exercises in this section.

4.3 More Examples

4.3.1. $\mathtt{0} \cup \mathtt{1} \cup \mathtt{0}\Sigma^*\mathtt{0} \cup \mathtt{1}\Sigma^*\mathtt{1}$.

4.3.2.

a) $\mathtt{0}\Sigma^*\mathtt{1}$.

b) $\Sigma\mathtt{1}\Sigma^*$.

c) $\Sigma^{k-1}\mathtt{1}\Sigma^*$.

4.3.3.

a) $(\mathtt{00} \cup \mathtt{11})\Sigma^*$.

b) $\Sigma^*(\mathtt{00} \cup \mathtt{11})$.

c) $\Sigma^*\mathtt{1}\Sigma$.

d) $\Sigma^*\mathtt{1}\Sigma^{k-1}$.

4.3.4.

a) $\Sigma^*\mathtt{1}\Sigma^*$.

b) $\mathtt{0}^*\mathtt{1}\mathtt{0}^*$.

c) $\Sigma^*\mathtt{1}\Sigma^*\mathtt{1}\Sigma^*$.

d) $\mathtt{0}^* \cup \mathtt{0}^*\mathtt{1}\mathtt{0}^*$.

e) $(\Sigma^*\mathtt{1})^k\Sigma^*$.

4.3.5.

a) $\varepsilon \cup \mathtt{1}\Sigma^* \cup \Sigma^*\mathtt{0}$.

b) $\varepsilon \cup \Sigma \cup \Sigma\mathtt{0}\Sigma^*$.

c) $(\varepsilon \cup \Sigma)^{k-1} \cup \Sigma^{k-1} 0 \Sigma^*$. Another solution: $(\cup_{i=0}^{k-1} \Sigma^i) \cup \Sigma^{k-1} 0 \Sigma^*$.

4.3.6.

a) $01\Sigma^* \cup 11\Sigma^* 0 \Sigma^*$.

b) $\Sigma^* 1 \Sigma^* 1 \Sigma^* \cup \Sigma^* 0 \Sigma^* 0 \Sigma^*$.

c) One way to go about this is to focus on the first two 1's that occur in the string and then list the ways in which the 0 in the string can relate to those two 0's. Here's what you get:

$$11^+ \cup 011^+ \cup 101^+ \cup 11^+ 01^*.$$

d) Let $E_0 = (1^*01^*0)^*1^*$ and $D_0 = (1^*01^*0)^*1^*01^*$. The regular expression E_0 describes the language of strings with an even number of 0's while D_0 describes the language of strings with an odd number of 0's. Then the language of strings that contain at least two 1's and an even number of 0's can be described as follows:

$$E_0 1 E_0 1 E_0 \cup E_0 1 D_0 1 D_0 \cup D_0 1 E_0 1 D_0 \cup D_0 1 D_0 1 E_0.$$

4.3.7.

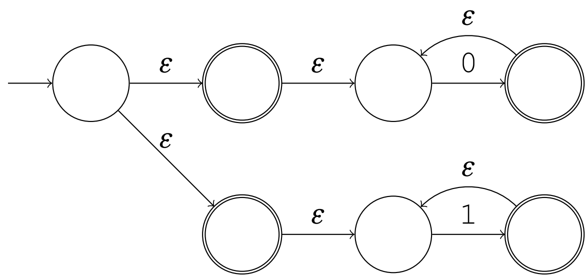
a) $(00)^* \# 11^+$.

b) $(00)^* 11^+$.

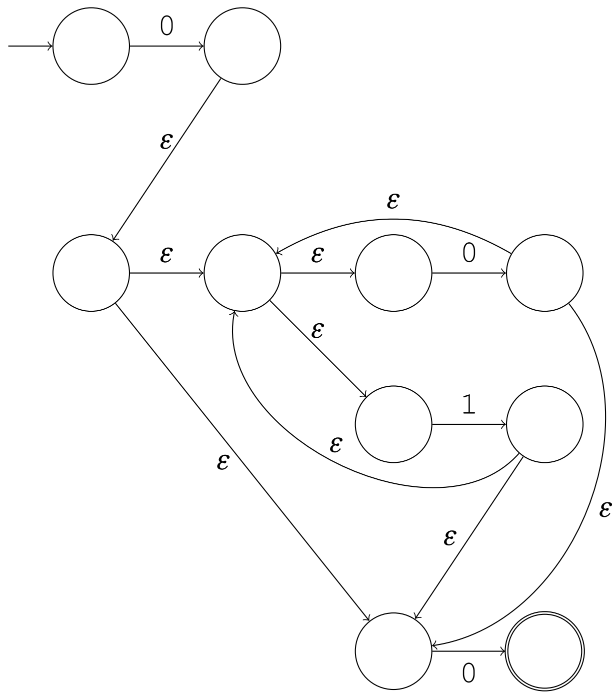
4.4 Converting Regular Expressions into DFA's

4.4.1.

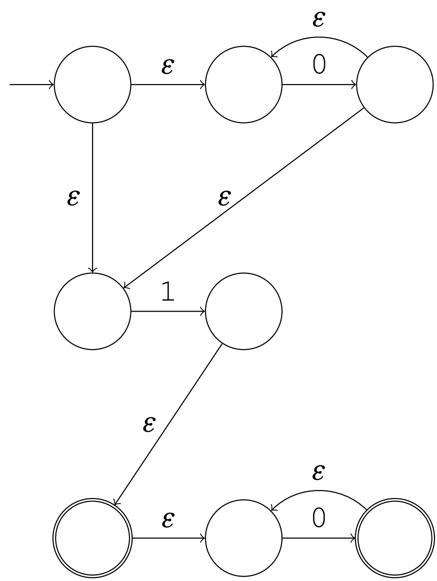
a)



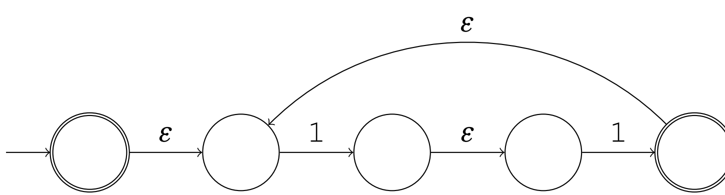
b)



c)

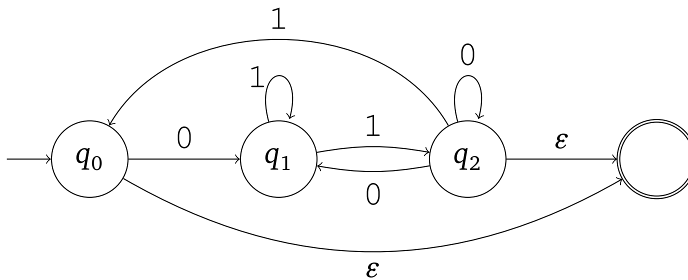


d)

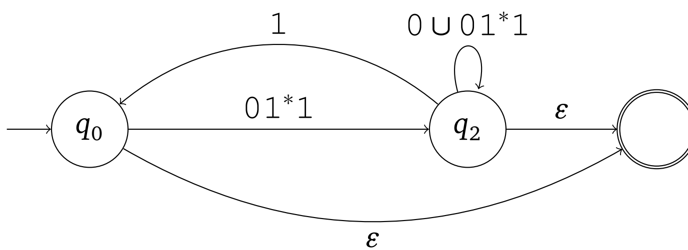


4.5 Converting DFA's into Regular Expressions

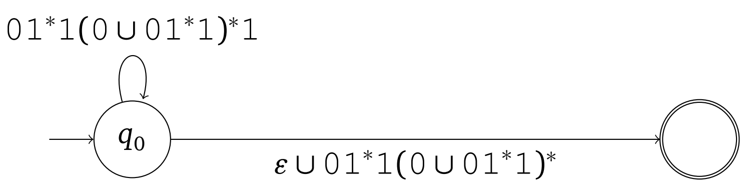
4.5.1. a) We first add a new accepting state:



We then remove state q_1:



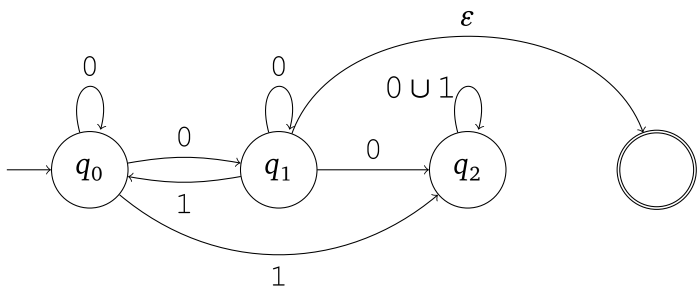
We remove state q_2:



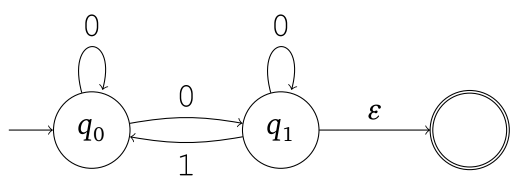
The final regular expression is

$$(01^*1(0 \cup 01^*1)^*1)^*(\varepsilon \cup 01^*1(0 \cup 01^*1)^*)$$

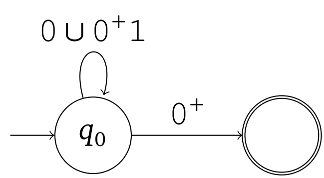
b) First, we add a new accepting state:



Then, we notice that state q_2 cannot be used to travel between the other two states. So we can just remove it:



We remove state q_1:



The final regular expression is $(0 \cup 0^+1)^*0^+$.

4.6 Precise Description of the Algorithm

4.6.1. Label the new accepting state q_4. Then the GNFA is $(Q, \{\mathtt{0}, \mathtt{1}\}, \delta, q_0, F)$ where

$$Q = \{q_0, q_2, q_3, q_4\}$$
$$F = \{q_4\}$$

and δ is defined by the following table:

δ	q_0	q_2	q_3	q_4
q_0	$\mathtt{0} \cup \mathtt{1}$	$\mathtt{00}$	$\emptyset$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\mathtt{1}$	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\mathtt{0} \cup \mathtt{1}$	ε
q_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

4.6.2. The GNFA is $(Q, \{\texttt{0}, \texttt{1}, \texttt{2}\}, \delta, q_0, F)$ where

$$Q = \{q_0, q_2, q_3\}$$
$$F = \{q_3\}$$

and δ is defined by the following table:

δ	q_0	q_2	q_3
q_0	$\texttt{0} \cup \texttt{1}\texttt{0}^*\texttt{2}$	$\texttt{2} \cup \texttt{1}\texttt{0}^*\texttt{1}$	$\varepsilon \cup \texttt{1}\texttt{0}^*$
q_2	$\texttt{1} \cup \texttt{2}\texttt{0}^*\texttt{2}$	$\texttt{0} \cup \texttt{2}\texttt{0}^*\texttt{1}$	$\texttt{2}\texttt{0}^*$
q_3	$\emptyset$	$\emptyset$	$\emptyset$

Chapter 5

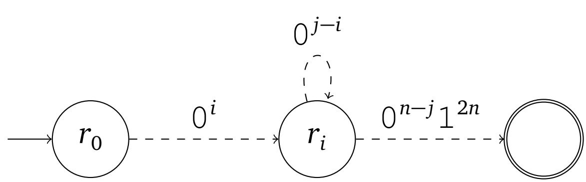
Nonregular Languages

5.1 Some Examples

5.1.1. Suppose that $L = \{0^n 1^{2n} \mid n \geq 0\}$ is regular. Let M be a DFA that recognizes L and let n be the number of states of M.

Consider the string $w = 0^n 1^{2n}$. As M reads the 0's in w, M goes through a sequence of states $r_0, r_1, r_2, \ldots, r_n$. Because this sequence is of length $n+1$, there must be a repetition in the sequence.

Suppose that $r_i = r_j$ with $i < j$. Then the computation of M on w looks like this:

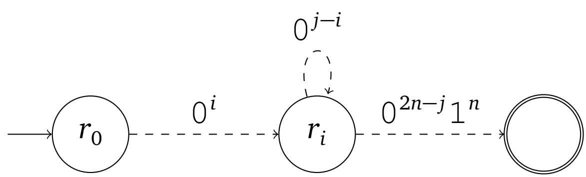


This implies that the string $0^i 0^{n-j} 1^{2n} = 0^{n-(j-i)} 1^{2n}$ is also accepted. But since this string no longer has exactly n $\mathtt{0}$'s, it cannot belong to L. This contradicts the fact that M recognizes L. Therefore, M cannot exist and L is not regular.

5.1.2. Suppose that $L = \{\mathtt{0}^i\mathtt{1}^j \mid 0 \leq i \leq 2j\}$ is regular. Let M be a DFA that recognizes L and let n be the number of states of M.

Consider the string $w = \mathtt{0}^{2n}\mathtt{1}^n$. As M reads the first n $\mathtt{0}$'s in w, M goes through a sequence of states $r_0, r_1, r_2, \ldots, r_n$. Because this sequence is of length $n+1$, there must be a repetition in the sequence.

Suppose that $r_i = r_j$ with $i < j$. Then the computation of M on w looks like this:

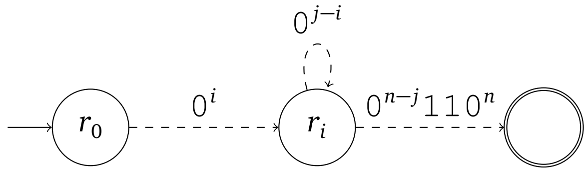


Now consider going twice around the loop. This implies that the string $\mathtt{0}^i\mathtt{0}^{2(j-i)}\mathtt{0}^{2n-j}\mathtt{1}^n = \mathtt{0}^{2n+(j-i)}\mathtt{1}^n$ is also accepted. But since this string has more than $2n$ $\mathtt{0}$'s, it does not belong to L. This contradicts the fact that M recognizes L. Therefore, M cannot exist and L is not regular.

5.1.3. Suppose that $L = \{ww^{\mathscr{R}} \mid w \in \{\mathtt{0},\mathtt{1}\}^*\}$ is regular. Let M be a DFA that recognizes L and let n be the number of states of M.

Consider the string $w = \mathtt{0}^n\mathtt{110}^n$. As M reads the first n $\mathtt{0}$'s of w, M goes through a sequence of states $r_0, r_1, r_2, \ldots, r_n$. Because this sequence is of length $n+1$, there must be a repetition in the sequence.

Suppose that $r_i = r_j$ with $i < j$. Then the computation of M on w looks like this:



This implies that the string $\texttt{0}^i\texttt{0}^{n-j}\texttt{110}^n = \texttt{0}^{n-(j-i)}\texttt{110}^n$ is also accepted. But this string does not belong to L. This contradicts the fact that M recognizes L. Therefore, M cannot exist and L is not regular.

5.2 The Pumping Lemma

5.2.1. Let $L = \{\texttt{0}^i\texttt{1}^j \mid i \leq j\}$. Suppose that L is regular. Let p be the pumping length. Consider the string $w = \texttt{0}^p\texttt{1}^p$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, w can be written as xyz where

1. $|xy| \leq p$.
2. $y \neq \varepsilon$.
3. $xy^kz \in L$, for every $k \geq 0$.

Condition (1) implies that y contains only $\texttt{0}$'s. Condition (2) implies that y contains at least one $\texttt{0}$. Therefore, the string xy^2z does not belong to L because it contains more $\texttt{0}$'s than $\texttt{1}$'s. This contradicts Condition (3) and implies that L is not regular.

5.2.2. Let $L = \{\texttt{1}^i\texttt{\#1}^j\texttt{\#1}^{i+j}\}$. Suppose that L is regular. Let p be the pumping length. Consider the string $w = \texttt{1}^p\texttt{\#1}^p\texttt{\#1}^{2p}$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, w can be written as xyz where

1. $|xy| \leq p$.
2. $y \neq \varepsilon$.
3. $xy^kz \in L$, for every $k \geq 0$.

Since $|xy| \leq p$, we have that y contains only 1's from the first part of the string. Therefore, $xy^2z = 1^{p+|y|}\#1^p\#1^{2p}$. Because $|y| \geq 1$, this string cannot belong to L. This contradicts the Pumping Lemma and shows that L is not regular.

5.2.3. Let L be the language described in the exercise. Suppose that L is regular. Let p be the pumping length. Consider the string $w = 1^p0\#1\#1^{p+1}$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, w can be written as xyz where

1. $|xy| \leq p$.
2. $y \neq \varepsilon$.
3. $xy^kz \in L$, for every $k \geq 0$.

Since $|xy| \leq p$, we have that y contains only 1's from the first part of the string. Therefore, $xy^2z = 1^{p+|y|}0\#1\#1^{p+1}$. Since $|y| \geq 1$, this string does not belong to L because the sum of the first two numbers no longer equals the third. This contradicts the Pumping Lemma and shows that L is not regular.

5.2.4. What is wrong with this proof is that we cannot assume that $p = 1$. All that the Pumping Lemma says is that p is positive. We cannot assume anything else about p. For example, if we get a contradiction for the case $p = 1$, then we haven't really contradicted the Pumping Lemma because it may be that p has another value.

Chapter 6

Context-Free Languages

6.1 Introduction

6.1.6.

a)

$$
\begin{aligned}
I &\to SN \\
S &\to - \mid \varepsilon \\
N &\to DN \mid D \\
D &\to 0 \mid \cdots \mid 9
\end{aligned}
$$

b)

$$\begin{aligned}
R &\to SN_1 \mid SN_0\,\texttt{.}\,N_1 \\
S &\to \texttt{-} \mid \varepsilon \\
N_0 &\to DN_0 \mid \varepsilon \\
N_1 &\to DN_0 \\
D &\to \texttt{0} \mid \cdots \mid \texttt{9}
\end{aligned}$$

c)

$$\begin{aligned}
I &\to \texttt{_}R_1 \mid LR_0 \\
R_0 &\to \texttt{_}R_0 \mid LR_0 \mid DR_0 \mid \varepsilon \\
R_1 &\to R_0LR_0 \mid R_0DR_0 \\
L &\to \texttt{a} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \cdots \mid \texttt{Z} \\
D &\to \texttt{0} \mid \cdots \mid \texttt{9}
\end{aligned}$$

6.2 Formal Definition of CFG's

There are no exercises in this section.

6.3 More Examples

6.3.1.

a)

$$S \to 0S0 \mid 1$$

b)

$$S \to 0S0 \mid 1S1 \mid \varepsilon$$

c)

$$S \to 0S11 \mid \varepsilon$$

d) Here's one solution:

$$\begin{aligned} S &\to ZS1 \mid \varepsilon \\ Z &\to 0 \mid \varepsilon \end{aligned}$$

Here's another one:

$$\begin{aligned} S &\to 0S1 \mid T \\ T &\to T1 \mid \varepsilon \end{aligned}$$

6.3.2.

$$S \to \mathtt{1}S\mathtt{1} \mid \#T$$
$$T \to \mathtt{1}T\mathtt{1} \mid \#$$

6.3.3.

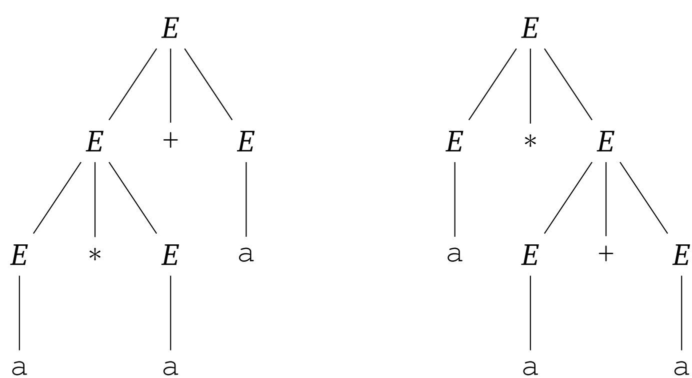
$$S \to \mathtt{(}S\mathtt{)}S \mid \mathtt{[}S\mathtt{]}S \mid \mathtt{\{}S\mathtt{\}}S \mid \varepsilon$$

6.3.4. A string of properly nested parentheses is either $\mathtt{()}$ or a string of the form $\mathtt{(}u\mathtt{)}v$ where u and v are either empty or strings of properly nested parentheses.

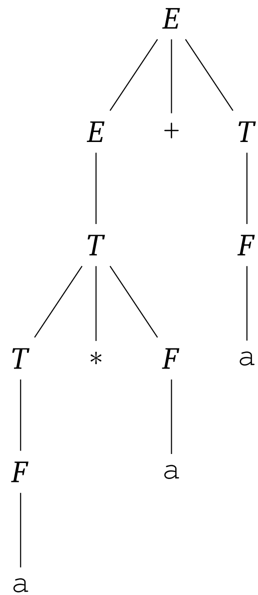
$$S \to \mathtt{(}S\mathtt{)}S \mid \mathtt{()}S \mid \mathtt{(}S\mathtt{)} \mid \mathtt{()}$$

6.4 Ambiguity and Parse Trees

6.4.4. Two parse trees in the first grammar:



The unique parse tree in the second grammar:



Chapter 7

Non Context-Free Languages

7.1 The Basic Idea

7.1.1. Let L denote that language and suppose that L is context-free. Let G be a CFG that generates L. Let $w = \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ where $n = b|V| + 1$. By the argument developed in this section, any derivation of w in G contains a repeated variable. Assume that the repetition is of the nested type. Then $uv^kxy^kz \in L$ for every $k \geq 0$. And, as shown at the end of the section, we can ensure that v and y are not both empty. There are now three cases to consider.

First, suppose that either v or y contains more than one type of symbol. Then $uv^2xy^2z \notin L$ because that string is not even in $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$.

Second, suppose that v and y each contain only one type of symbol but no c's. Then uv^2xy^2z contains more a's or b's than c's. Therefore, $uv^2xy^2z \notin L$.

Third, suppose that v and y each contain only one type of symbol including some $\mathtt{c}$'s. Then v and y cannot contain both $\mathtt{a}$'s or $\mathtt{b}$'s. This implies that uv^0xy^0z contains less $\mathtt{c}$'s than $\mathtt{a}$'s or less $\mathtt{c}$'s than $\mathtt{b}$'s. Therefore, $uv^0xy^0z \notin L$.

In all three cases, we have a contradiction. This proves that L is not context-free.

7.2 A Pumping Lemma

7.2.1. Let L denote that language and suppose that L is context-free. Let p be the pumping length. Consider the string $w = \mathtt{0}^p\mathtt{1}^p\mathtt{0}^p$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, w can be written as $uvxyz$ where

1. $vy \neq \varepsilon$.
2. $uv^kxy^kz \in L$, for every $k \geq 0$.

There are two cases to consider. First, suppose that either v or y contains more than one type of symbol. Then $uv^2xy^2z \notin L$ because that string is not even in $\mathtt{0}^*\mathtt{1}^*\mathtt{0}^*$.

Second, suppose that v and y each contain only one type of symbol. The string w consists of three blocks of p symbols and v and y can touch at most two of those blocks. Therefore, $uv^2xy^2z = \mathtt{0}^{p+i}\mathtt{1}^{p+j}\mathtt{0}^{p+k}$ where at least one of i, j, k is greater than 0 and at least one of i, j, k is equal to 0. This implies that $uv^2xy^2z \notin L$.

In both cases, we have that $uv^2xy^2z \notin L$. This is a contradiction and proves that L is not context-free.

7.2.2. Let L denote that language and suppose that L is context-free. Let p be the pumping length. Consider the string $w = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, w can be written as $uvxyz$ where

1. $vy \neq \varepsilon$.
2. $uv^kxy^kz \in L$, for every $k \geq 0$.

There are three cases to consider. First, suppose that either v or y contains more than one type of symbol. Then $uv^2xy^2z \notin L$ because that string is not even in $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$.

In the other two cases, v and y each contain only one type of symbol. The second case is when v consists of a's. Then, since y cannot contain both b's and c's, uv^2xy^2z contains more a's than b's or more a's than c's. This implies that $uv^2xy^2z \notin L$.

The third case is when v does not contain any a's. Then y can't either. This implies that uv^0xy^0z contains less b's than a's or less c's than a's. Therefore, $uv^0xy^0z \notin L$.

In all cases, we have that $uv^kxy^kz \notin L$ for some $k \geq 0$. This is a contradiction and proves that L is not context-free.

7.2.3. Let L denote that language and suppose that L is context-free. Let p be the pumping length. Consider the string $w = \mathtt{1}^p\#\mathtt{1}^p\#\mathtt{1}^{2p}$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, w can be written as $uvxyz$ where

1. $vy \neq \varepsilon$.
2. $uv^kxy^kz \in L$, for every $k \geq 0$.

There are several cases to consider. First, suppose that either v or y contains a #. Then $uv^2xy^2z \notin L$ because it contains too many #'s.

For the remaining cases, assume that neither v nor y contains a #. Note that w consists of three blocks of 1's separated by #'s. This implies that v and y are each completely contained within one block and that v and y cannot contain 1's from all three blocks.

The second case is when v and y don't contain any 1's from the third block. Then $uv^2xy^2z = 1^{p+i}\#1^{p+j}\#1^{2p}$ where at least one of i, j is greater than 0. This implies that $uv^2xy^2z \notin L$.

The third case is when v and y don't contain any 1's from the first two blocks. Then $uv^2xy^2z = 1^p\#1^p\#1^{2p+i}$ where $i > 0$. This implies that $uv^2xy^2z \notin L$.

The fourth case is when v consists of 1's the first block and y consists of 1's the third block. Then $uv^2xy^2z = 1^{p+i}\#1^p\#1^{2p+j}$ where both i, j are greater than 0. This implies that $uv^2xy^2z \notin L$ because the first block is larger than the second block.

The fifth and final case is when v consists of 1's the second block and y consists of 1's the third block. Then $uv^0xy^0z = 1^p\#1^{p-i}\#1^{2p-j}$ where both i, j are greater than 0. This implies that $uv^2xy^2z \notin L$ because the second block is smaller than the first block.

In all cases, we have a contradiction. This proves that L is not context-free.

7.3 A Stronger Pumping Lemma

7.3.1. Let L denote that language and suppose that L is context-free. Let p be the pumping length. Consider the string $w = 0^p1^{2p}0^p$. Clearly, $w \in L$ and

$|w| \geq p$. Therefore, according to the Pumping Lemma, w can be written as $uvxyz$ where

1. $|vxy| \leq p$.
2. $vy \neq \varepsilon$.
3. $uv^kxy^kz \in L$, for every $k \geq 0$.

The string w consists of three blocks of symbols. Since $|vxy| \leq p$, v and y are completely contained within two consecutive blocks. Suppose that v and y are both contained within a single block. Then uv^2xy^2z has additional symbols of one type but not the other. Therefore, this string is not in L.

Now suppose that v and y touch two consecutive blocks, the first two, for example. Then $uv^0xy^0z = \mathtt{0}^i\mathtt{1}^j\mathtt{0}^p\mathtt{1}^p$ where $0 < i, j < p$. This string is clearly not in L. The same is true for the other blocks.

Therefore, in all cases, we have that w cannot be pumped. This contradicts the Pumping Lemma and proves that L is not context-free.

7.3.2. Let L denote that language and suppose that L is context-free. Let p be the pumping length. Consider the string $w = \mathtt{1}^p\#\mathtt{1}^p\#\mathtt{1}^{p^2}$. Clearly, $w \in L$ and $|w| \geq p$. Therefore, according to the Pumping Lemma, w can be written as $uvxyz$ where

1. $vy \neq \varepsilon$.
2. $uv^kxy^kz \in L$, for every $k \geq 0$.

There are several cases to consider. First, suppose that either v or y contains a $\#$. Then $uv^2xy^2z \notin L$ because it contains too many #'s.

For the remaining cases, assume that neither v or y contains a `#`. Note that w consists of three blocks of `1`'s separated by `#`'s. This implies that v and y are each completely contained within one block and that v and y cannot touch all three blocks.

The second case is when v and y are contained within the first two blocks. Then $uv^2xy^2z = 1^{p+i}\#1^{p+j}\#1^{p^2}$ where at least one of i, j is greater than 0. This implies that $uv^2xy^2z \notin L$.

The third case is when v and y are both within the third block. Then $uv^2xy^2z = 1^p\#1^p\#1^{p^2+i}$ where $i > 0$. This implies that $uv^2xy^2z \notin L$.

The fourth case is when v consists of `1`'s from the first block and y consists of `1`'s from the third block. This case cannot occur since $|vxy| \leq p$.

The fifth and final case is when v consists of `1`'s from the second block and y consists of `1`'s from the third block. Then $uv^2xy^2z = 1^p\#1^{p+i}\#1^{p^2+j}$ where both i, j are greater than 0. Now, $p(p+i) \geq p(p+1) = p^2 + p$. On the other hand, $p^2 + j < p^2 + p$ since $j = |y| < |vxy| \leq p$. Therefore, $p(p+i) \neq p^2 + j$. This implies that $uv^2xy^2z \notin L$.

In all cases, we have a contradiction. This proves that L is not context-free.

Chapter 8

More on Context-Free Languages

8.1 Closure Properties

8.1.2. Here's a CFG for the language $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i \neq j \text{ or } j \neq k\}$:

$$\begin{aligned}
S &\to TC_0 \mid A_0U \\
T &\to \mathtt{a}T\mathtt{b} \mid A_1 \mid B_1 \quad (\mathtt{a}^i\mathtt{b}^j,\ i \neq j) \\
U &\to \mathtt{b}U\mathtt{c} \mid B_1 \mid C_1 \quad (\mathtt{b}^j\mathtt{c}^k,\ j \neq k) \\
A_0 &\to \mathtt{a}A_0 \mid \varepsilon \quad (\mathtt{a}^*) \\
C_0 &\to \mathtt{c}C_0 \mid \varepsilon \quad (\mathtt{c}^*) \\
A_1 &\to \mathtt{a}A_1 \mid \mathtt{a} \quad (\mathtt{a}^+) \\
B_1 &\to \mathtt{b}B_1 \mid \mathtt{b} \quad (\mathtt{b}^+) \\
C_1 &\to \mathtt{c}C_1 \mid \mathtt{c} \quad (\mathtt{c}^+)
\end{aligned}$$

Now, the complement of $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is

$$\overline{\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*} \cup \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i \neq j \text{ or } j \neq k\}.$$

The language on the left is regular and, therefore, context-free. We have just shown that the language on the right is context-free. Therefore, the complement of $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is context-free because the union of two CFL's is always context-free.

8.1.3. Suppose that $w = xy$ where $|x| = |y|$ but $x \neq y$. Focus on one of the positions where x and y differ. It must be the case that $x = u_1 a u_2$ and $y = v_1 b v_2$, where $|u_1| = |v_1|$, $|u_2| = |v_2|$, $a, b \in \{\mathtt{0}, \mathtt{1}\}$ and $a \neq b$. This implies that $w = u_1 a u_2 v_1 b v_2$. Now, notice that $|u_2 v_1| = |v_2| + |u_1|$. We can then split $u_2 v_1$ differently, as $s_1 s_2$ where $|s_1| = |u_1|$ and $|s_2| = |v_2|$. This implies that $w = u_1 a s_1 s_2 b v_2$ where $|u_1| = |s_1|$ and $|s_2| = |v_2|$. The idea behind a CFG that derives w is to generate $u_1 a s_1$ followed by $s_2 b v_2$. Here's the result:

$$\begin{aligned}
S &\to T_0 T_1 \mid T_1 T_0 \\
T_0 &\to A T_0 A \mid \mathtt{0} \quad (u\mathtt{0}s,\ |u| = |s|) \\
T_1 &\to A T_1 A \mid \mathtt{1} \quad (u\mathtt{1}s,\ |u| = |s|) \\
A &\to \mathtt{0} \mid \mathtt{1}
\end{aligned}$$

Now, the complement of $\{ww \mid w \in \{\mathtt{0}, \mathtt{1}\}^*\}$ is

$$\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid |w| \text{ is odd}\} \cup \{xy \mid x, y \in \{\mathtt{0}, \mathtt{1}\}^*, |x| = |y| \text{ but } x \neq y\}$$

The language on the left is regular and, therefore, context-free. We have just shown that the language on the right is context-free. Therefore, the complement of $\{ww \mid w \in \{0,1\}^*\}$ is context-free because the union of two CFL's is always context-free.

8.2 Pushdown Automata

8.2.2. One possible solution is to start with a CFG for this language and then simulate this CFG with a stack algorithm. Here's a CFG for this language:

$$S \to 0S1$$
$$S \to \varepsilon$$

Now, here's a single-scan stack algorithm that simulates this CFG:

```
push S on the stack
while (stack not empty)
    if (top of stack is S)
        nondeterministically choose to replace S
            by 0S1 (with 0 at the top of the
            stack) or to delete S
    else // top of stack is 0 or 1
        if (end of input) reject
        read next input symbol c
        if (c equals top of stack)
            pop stack
        else
            reject
if (end of input)
    accept
else
    reject
```

Another solution is a more direct algorithm:

```
if (end of input) accept
initialize stack to empty
read next char c
while (c is 0)
    push 0 on the stack
    if (end of input) // some 0's but no 1's
        reject
    read next char c
while (c is 1)
    if (stack empty) reject  // too many 1's
    pop stack
    if (end of input)
        if (stack empty)
            accept
        else
            reject  // too many 0's
    read next char c
reject  // 0's after 1's
```

8.3 Deterministic Algorithms for CFL's

8.3.3. Let L be the language of strings of the form ww. We know that L is not context-free. If $\overline{L}$ was a DCFL, then L would be also be a DCFL because that class is closed under complementation. This would contradict the fact that L is not even context-free.

Chapter 9

Turing Machines

9.1 Introduction

9.1.1. The idea is to repeatedly cross off one a, one b and one c.

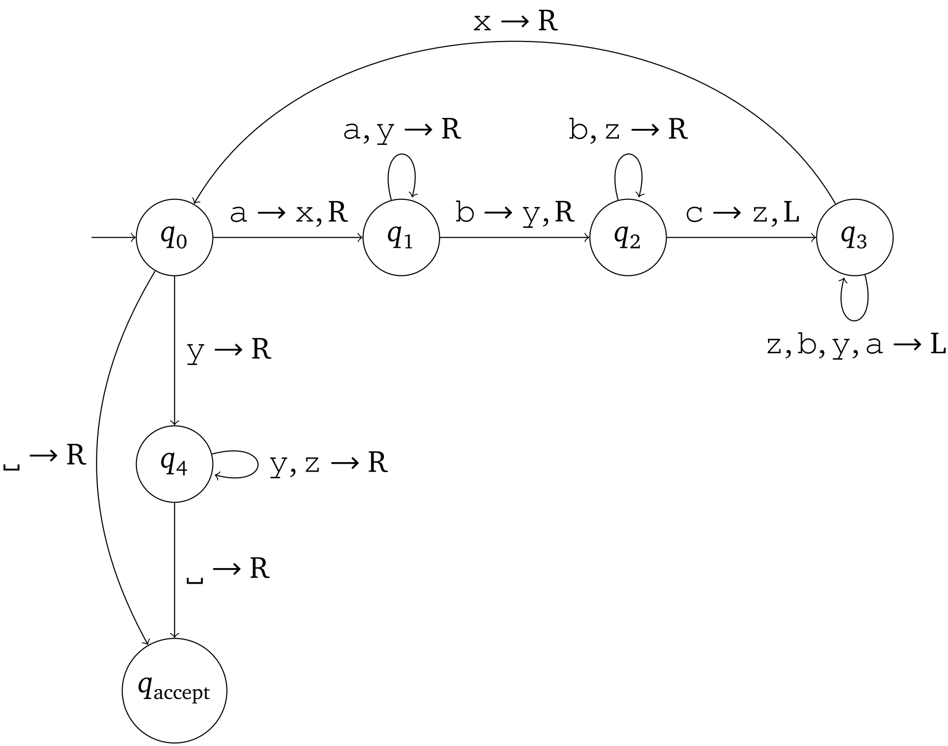
1. If the input is empty, accept.
2. Scan the input to verify that it is of the form $a^*b^*c^*$. If not, reject.
3. Return the head to the beginning of the memory.
4. Cross off the first a.
5. Move right to the first b and cross it off. If no b can be found, reject.
6. Move right to the first c and cross it off. If no c can be found, reject.
7. Repeat Steps 2 to 5 until all the a's have been crossed off. When that happens, scan right to verify that all other symbols have been crossed off. If so, accept. Otherwise, reject.

9.1.2. The idea is to first find the middle of the string and then proceed as we did in this section for the language $\{w\#w\}$. In what follows, when we mark a symbol with an L, we change it into either $\texttt{0}^{\mathrm{L}}$ or $\texttt{1}^{\mathrm{L}}$. Similarly for marking with an R.

1. If the input is empty, accept.
2. Mark the first unmarked symbol with an L.
3. Move right to the last unmarked symbol. If none can be found, reject (because the input is of odd length). Otherwise, mark it with an R and move left.
4. Repeat Steps 2 and 3 until all the symbols have been marked. The input is now of the form uv where $|u| = |v|$, all the symbols of u are marked with an L and all the symbols of v are marked with an R.
5. Verify that $u = v$ by following Steps 2 to 5 of the TM for the language $\{w\#w\}$.

9.2 Formal Definition

9.2.2. We will cross off an `a` by replacing it with an `x`. For `b` and `c`, we will use `y` and `z`, respectively. Missing transitions go to the rejecting state.



9.3 Variations on the Basic Turing Machine

9.3.2. Suppose that M is a Turing machine with doubly infinite memory. We construct a basic Turing machine M' that simulates M as follows.

1. Let w be the input string. Shift w one position to the right. Place a # before and after w so the tape contains $\#w\#$.

2. Move the head to the first symbol of w and run M.

3. Whenever M moves to the rightmost #, replace it with a blank and write a # in the next position. Return to the blank and continue running M.

4. Whenever M moves to the leftmost #, shift the entire contents of the memory (up to the rightmost #) one position to the right. Write a # and a blank in the first two positions, put the head on that blank and continue running M.

5. Repeat Steps 2 to 4 until M halts. Accept if M accepts. Otherwise, reject.

9.3.3. Suppose that L_1 and L_2 are decidable languages. Let M_1 and M_2 be TM's that decide these languages. Here's a TM that decides $\overline{L_1}$:

1. Run M_1 on the input.

2. If M_1 accepts, reject. If M_1 rejects, accept.

Here's a TM that decides $L_1 \cup L_2$:

1. Copy the input to a second tape.

2. Run M_1 on the first tape.

3. If M_1 accepts, accept.

4. Otherwise, run M_2 on the second tape.

5. If M_2 accepts, accept. Otherwise, reject,

Here's a TM that decides $L_1 \cap L_2$:

1. Copy the input to a second tape.
2. Run M_1 on the first tape.
3. If M_1 rejects, reject.
4. Otherwise, run M_2 on the second tape.
5. If M_2 accepts, accept. Otherwise, reject,

Here's a TM that decides $L_1 L_2$:

1. If the input is empty, run M_1 on the first tape and M_2 on a blank second tape. If both accept, accept. Otherwise, reject.
2. Mark the first symbol of the input. (With an underline, for example.)
3. Copy the beginning of the input, up to but *not* including the marked symbol, to a second tape. Copy the rest of the input to a third tape.
4. Run M_1 on the second tape and M_2 on the third tape.
5. If both accept, accept.
6. Otherwise, move the mark to the next symbol of the input.
7. While the mark has not reached a blank space, repeat Steps 3 to 6.
8. Delete the mark from the first tape. Run M_1 on the first tape and M_2 on a blank second tape. If both accept, accept. Otherwise, reject.

9.3.4.

1. Verify that the input is of the form $x\#y\#z$ where x, y and z are strings of digits of the same length. If not, reject.

2. Write a # in the first position of tapes 2, 3 and 4.

3. Copy x, y and z to tapes 2, 3 and 4, respectively.

4. Set the carry to 0. (Remember the carry with the states of the TM.)

5. Scan those numbers simultaneously from right to left, using the initial # to know when to stop. For each position, compute the sum n of the carry and the digits of x and y (using the transition function). If $n \bmod 10$ is not equal to the digit of z, reject. Set the carry to $\lfloor n/10 \rfloor$.

6. If the carry is 0, accept. Otherwise, reject.

9.4 Equivalence with Programs

9.4.1. Here's a TM for the copy instruction:

1. Move the memory head to location i.

2. Copy 32 bits starting at that memory location to an extra tape.

3. Move the memory head to location j.

4. Copy the 32 bits from the extra tape to the 32 bits that start at the current memory location.

Here's a TM for the add instruction:

1. Move the memory head to location i.

2. Copy 32 bits starting at that memory location to a second extra tape.

3. Move the memory head to location j.

4. Add the 32 bits from the second extra tape to the 32 bits that start at the current memory location. (This can be done by adapting the solution to an exercise from the previous section.) Discard any leftover carry.

Here's a TM for the jump-if instruction:

1. Move the memory head to location i.
2. Scan the 32 bits that start at that memory location. If they're all 0, transition to the first state of the group of states that implement the other instruction. Otherwise, continue to the next instruction.

9.4.2. Suppose that L is a decidable language. Let M be a TM that decides this language. Here's a TM that decides L^*. (Note that this is a high-level description.)

```
Let w be the input and let n be the length of w
If w is empty, accept
For each k in {1, 2, ..., n}
   For every partition of w into k substrings
         s[1], ..., s[k]
      Run M on each s[i]
      If M accepts all of them, accept
Reject
```

Chapter 10

Problems Concerning Formal Languages

10.1 Regular Languages

10.1.1. If M is a DFA with input alphabet Σ, then $L(M) = \Sigma^*$ if and only if $\overline{L(M)} = \emptyset$. This leads to the following algorithm for $\mathrm{ALL}_{\mathrm{DFA}}$:

1. Verify that the input string is of the form $\langle M \rangle$ where M is a DFA. If not, reject.
2. Construct a DFA M' for the complement of $L(M)$. (This can be done by simply switching the acceptance status of every state in M.)
3. Test if $L(M') = \emptyset$ by using the emptiness algorithm.
4. Accept if that algorithm accepts. Otherwise, reject.

10.1.2. The key observation here is that $L(R_1) \subseteq L(R_2)$ if and only if $L(R_1) - L(R_2) = \emptyset$. Since $L(R_1) - L(R_2) = L(R_1) \cap \overline{L(R_2)}$, this language is regular.

This leads to the following algorithm for $\text{SUBSET}_{\text{REX}}$:

1. Verify that the input string is of the form $\langle R_1, R_2 \rangle$ where R_1 and R_2 are regular expressions. If not, reject.
2. Construct a DFA M for the language $L(R_1) - L(R_2)$. (This can be done by converting R_1 and R_2 to DFA's and then combining these DFA's using the constructions for closure under complementation and intersection.)
3. Test if $L(M) = \emptyset$ by using the emptiness algorithm.
4. Accept if that algorithm accepts. Otherwise, reject.

10.1.3. Let L_{ODD} denote the language of strings of odd length. If M is a DFA, then M accepts at least one string of odd length if and only if $L(M) \cap L_{\text{ODD}} \neq \emptyset$. Since L_{ODD} is regular, this leads to the following algorithm:

1. Verify that the input string is of the form $\langle M \rangle$ where M is a DFA. If not, reject.
2. Construct a DFA M' for the language $L(M) \cap L_{\text{ODD}}$. (This can be done by combining M with a DFA for L_{ODD} using the construction for closure under interesction.)
3. Test if $L(M') = \emptyset$ by using the emptiness algorithm.
4. Reject if that algorithm accepts. Otherwise, accept.

10.2 CFL's

10.2.1. Here's an algorithm:

1. Verify that the input string is of the form $\langle G \rangle$ where G is a CFG. If not, reject.
2. Determine if G derives ε by using an algorithm for A_{CFG}.
3. Accept if that algorithm accepts. Otherwise, reject.

10.2.2. Let L_{ODD} denote the language of strings of odd length. If G is a CFG, then G derives at least one string of odd length if and only if $L(G) \cap L_{\text{ODD}} \neq \emptyset$. This leads to the following algorithm:

1. Verify that the input string is of the form $\langle G \rangle$ where G is a CFG. If not, reject.
2. Construct a CFG G' for the language $L(G) \cap L_{\text{ODD}}$. (This can be done by converting G into a PDA, combining it with a DFA for L_{ODD} as outlined in Section 8.2, and converting the resulting PDA into a CFG.)
3. Test if $L(G') = \emptyset$ by using the emptiness algorithm.
4. Reject if that algorithm accepts. Otherwise, accept.

Chapter 11

Undecidability

11.1 An Unrecognizable Language

11.1.1. Suppose, by contradiction, that L is recognized by some Turing machine M. In other words, for every string w, M accepts w if and only if $w \in L$. In particular,

$$M \text{ accepts } \langle M \rangle \text{ if and only if } \langle M \rangle \in L$$

But the definition of L tells us that

$$\langle M \rangle \in L \text{ if and only if } M \text{ does not accept } \langle M \rangle$$

This is a contradiction. Therefore, M cannot exist and L is not recognizable.

11.2 Natural Undecidable Languages

11.2.1. Here's a Turing machine that recognizes D:

1. Let w be the input string. Find i such that $w = s_i$.
2. Generate the encoding of machine M_i.
3. Simulate M_i on s_i.
4. If M accepts, accept. Otherwise, reject.

11.3 Reducibility and Additional Examples

11.3.1. Suppose that algorithm R decides $\text{BUMPS\_OFF\_LEFT}_{\text{TM}}$. We use this algorithm to design an algorithm S for the acceptance problem:

1. Verify that the input string is of the form $\langle M, w \rangle$ where M is a Turing machine and w is a string over the input alphabet of M. If not, reject.
2. Without loss of generality, suppose that # is a symbol not in the tape alphabet of M. (Otherwise, pick some other symbol.) Construct the following Turing machine M':
   (a) Let x be the input string. Shift x one position to the right. Place a # before x so the tape contains $\#x$.
   (b) Move the head to the first symbol of x and run M.
   (c) Whenever M moves the head to the #, move the head back one position to the right.
   (d) If M accepts, move the head to the # and move left again.
3. Run R on $\langle M', w \rangle$.

4. If R accepts, accept. Otherwise, reject.

To prove that S decides $\mathrm{A_{TM}}$, first suppose that M accepts w. Then when M' runs on w, it attempts to move left from the first position of its tape (where the # is). This implies that R accepts $\langle M', w \rangle$ and that S accepts $\langle M, w \rangle$, which is what we want.

Second, suppose that M does not accept w. Then when M' runs on w, it never attempts to move left from the first position of its tape. This implies that R rejects $\langle M', w \rangle$ and that S rejects $\langle M, w \rangle$. Therefore, S decides $\mathrm{A_{TM}}$.

Since $\mathrm{A_{TM}}$ is undecidable, this is a contradiction. Therefore, R does not exist and $\mathrm{BUMPS\_OFF\_LEFT_{TM}}$ is undecidable.

11.3.2. Suppose that algorithm R decides $\mathrm{ENTERS\_STATE_{TM}}$. We use this algorithm to design an algorithm S for the acceptance problem:

1. Verify that the input string is of the form $\langle M, w \rangle$ where M is a Turing machine and w is a string over the input alphabet of M. If not, reject.
2. Run R on $\langle M, w, q_{\mathrm{accept}} \rangle$, where q_{accept} is the accepting state of M.
3. If R accepts, accept. Otherwise, reject.

It's easy to see that S decides $\mathrm{A_{TM}}$ because M accepts w if and only if M enters its accepting state while running on w.

Since $\mathrm{A_{TM}}$ is undecidable, this is a contradiction. Therefore, R does not exist and $\mathrm{ENTERS\_STATE_{TM}}$ is undecidable.

11.3.4. Suppose that algorithm R decides $\mathrm{ACCEPTS}\varepsilon_{\mathrm{TM}}$. We use this algorithm to design an algorithm S for the acceptance problem:

1. Verify that the input string is of the form $\langle M, w \rangle$ where M is a Turing machine and w is a string over the input alphabet of M. If not, reject.

2. Construct the following Turing machine M':
   (a) Let x be the input string. If $x \neq \varepsilon$, reject.
   (b) Run M on w.
   (c) If M accepts, accept. Otherwise, reject.
3. Run R on $\langle M' \rangle$.
4. If R accepts, accept. Otherwise, reject.

To prove that S decides A_{TM}, first suppose that M accepts w. Then $L(M') = \{\varepsilon\}$, which implies that R accepts $\langle M' \rangle$ and that S accepts $\langle M, w \rangle$, which is what we want. On the other hand, suppose that M does not accept w. Then $L(M') = \emptyset$, which implies that R rejects $\langle M' \rangle$ and that S rejects $\langle M, w \rangle$, which is again what we want.

Since A_{TM} is undecidable, this is a contradiction. Therefore, R does not exist and $\text{ACCEPTS}\varepsilon_{TM}$ is undecidable.

11.3.5. Suppose that algorithm R decides HALTS_ON_ALL_{TM}. We use this algorithm to design an algorithm S for HALT_{TM}:

1. Verify that the input string is of the form $\langle M, w \rangle$ where M is a Turing machine and w is a string over the input alphabet of M. If not, reject.
2. Construct the following Turing machine M':
   (a) Run M on w.
   (b) If M accepts, accept. Otherwise, reject.
3. Run R on $\langle M' \rangle$.
4. If R accepts, accept. Otherwise, reject.

Note that M' ignores its input and always runs M on w. This implies that if M halts on w, then M' halts on every input, R accepts M' and S accepts $\langle M, w\rangle$. On the other hand, if M doesn't halt on w, then M' doesn't halts on any input, R rejects M' and S rejects $\langle M, w\rangle$. This shows that S decides $\mathrm{HALT}_{\mathrm{TM}}$.

However, we know that $\mathrm{HALT}_{\mathrm{TM}}$ is undecidable. So this is a contradiction, which implies that R does not exist and $\mathrm{HALTS\_ON\_ALL}_{\mathrm{TM}}$ is undecidable.

11.3.6. Suppose that algorithm R decides $\mathrm{EQ}_{\mathrm{CFG}}$. We use this algorithm to design an algorithm S for $\mathrm{ALL}_{\mathrm{CFG}}$:

1. Verify that the input string is of the form $\langle G\rangle$ where G is a CFG. If not, reject.
2. Let Σ be the alphabet of G and construct a CFG G' that generates Σ^*.
3. Run R on $\langle G, G'\rangle$.
4. If R accepts, accept. Otherwise, reject.

This algorithm accepts $\langle G\rangle$ if and only if $L(G) = L(G') = \Sigma^*$. Therefore, S decides $\mathrm{ALL}_{\mathrm{CFG}}$, which contradicts the fact that $\mathrm{ALL}_{\mathrm{CFG}}$. Therefore, R does not exist and $\mathrm{EQ}_{\mathrm{CFG}}$ is undecidable.

11.3.7. Yes, the proof of the theorem still works, as long as we make two changes.

First, Step 4 in the description of S should be changed to the following:

> If R accepts, reject. Otherwise, accept.

Second, the paragraph that follows the description of S should be changed as follows:

Suppose that M accepts w. Then $L(M') = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$, which implies that R rejects $\langle M' \rangle$ and that S accepts $\langle M, w \rangle$. On the other hand, suppose that M does not accept w. Then $L(M') = \emptyset$, which implies that R accepts $\langle M' \rangle$ and that S rejects $\langle M, w \rangle$. Therefore, S decides A_{TM}.

11.3.8. Suppose that algorithm R decides INFINITE_{TM}. We use this algorithm to design an algorithm S for A_{TM}:

1. Verify that the input string is of the form $\langle M, w \rangle$ where M is a Turing machine and w is a string over the input alphabet of M. If not, reject.
2. Construct the following Turing machine M':
   (a) Run M on w.
   (b) If M accepts, accept. Otherwise, reject.
3. Run R on $\langle M' \rangle$.
4. If R accepts, accept. Otherwise, reject.

To prove that S decides A_{TM}, first suppose that M accepts w. Then M' accepts every input string, which implies that $L(M')$ is infinite, R accepts M' and S accepts $\langle M, w \rangle$. On the other hand, if M doesn't accept w, then $L(M') = \emptyset$, R rejects M' and S rejects $\langle M, w \rangle$.

Since A_{TM} is undecidable, this is a contradiction. Therefore, R does not exist and INFINITE_{TM} is undecidable.

11.3.9. Suppose that algorithm R decides DECIDABLE_{TM}. We use this algorithm to design an algorithm S for the acceptance problem. Given a Turing machine M and a string w, S will construct a new Turing machine M' whose language is decidable if and only if M accepts w.

Let M_D be a Turing machine that recognizes the language D defined in Section 11.1. Exercise 11.2.1 asked you to show that M_D exists. Let Σ be the alphabet of D. (D can be defined over any alphabet.)

1. Verify that the input string is of the form $\langle M, w\rangle$ where M is a Turing machine and w is a string over the input alphabet of M. If not, reject.
2. Construct the following Turing machine M' with input alphabet Σ:
   (a) Let x be the input string.
   (b) Run M_D on x. If M_D accepts, accept.
   (c) Otherwise, run M on w.
   (d) If M accepts, accept. Otherwise, reject.
3. Run R on $\langle M'\rangle$.
4. If R accepts, accept. Otherwise, reject.

To show that S decides A_{TM}, first suppose that M accepts w. Then $L(M') = \Sigma^*$, which is a decidable language. This implies that R accepts $\langle M'\rangle$ and that S accepts $\langle M, w\rangle$. On the other hand, suppose that M does not accept w. Then $L(M') = D$, which is undecidable. This implies that R rejects $\langle M'\rangle$ and that S rejects $\langle M, w\rangle$.

Since A_{TM} is undecidable, this is a contradiction. Therefore, R does not exist and $\text{DECIDABLE}_{\text{TM}}$ is undecidable.

11.4 Rice's Theorem

11.4.1. Yes, because it is still true that $L(M') = L$ when M accepts w, and that $L(M') = \emptyset$ when M does not accept w.

11.5 Natural Unrecognizable Languages

11.5.1.

a) Suppose that L is recognizable and that B is decidable. Let M_L and M_B be Turing machines that recognize L and B, respectively. We use M_L and M_B to design a Turing machine M that recognizes $L \cup B$:

   1. Let w be the input string. Run M_B on w.
   2. If B accepts, accept.
   3. Otherwise (if B rejects), run M_L on w.
   4. If M_L accepts, accept. Otherwise, reject.

   (Note that the order in which M runs M_B and M_L is important. If the order was reversed, M might get stuck running M_L forever without getting a chance to run M_B. That would be a problem for strings in $B - L$.)

b) Suppose that L is recognizable and that B is decidable. Let M_L and M_B be Turing machines that recognize L and B, respectively. We use M_L and M_B to design a Turing machine M that recognizes $L \cap B$:

   1. Let w be the input string. Run M_B on w.
   2. If B rejects, reject.
   3. Otherwise (if B accepts), run M_L on w.
   4. If M_L accepts, accept. Otherwise, reject.

   (The order in which M runs M_B and M_L is again important.)

11.5.2. From an exercise in the previous section, we know that $\text{ACCEPTS}\varepsilon_{\text{TM}}$ is undecidable. The following Turing machine shows that $\text{ACCEPTS}\varepsilon_{\text{TM}}$ is recognizable:

1. Verify that the input string is of the form $\langle M \rangle$ where M is a Turing machine. If not, reject.

2. Simulate M on ε.

3. If M accepts, accept. Otherwise, reject.

Therefore, by the results of this section, $\overline{\text{ACCEPTS}\varepsilon}_{\text{TM}}$ is not recognizable.

Now, $\overline{\text{ACCEPTS}\varepsilon}_{\text{TM}}$ is the union of $\text{NACCEPTS}\varepsilon_{\text{TM}}$ and the set of strings that are not of the form $\langle M \rangle$ where M is a Turing machine. If $\text{NACCEPTS}\varepsilon_{\text{TM}}$ was recognizable, then $\overline{\text{ACCEPTS}\varepsilon}_{\text{TM}}$ would be too. Therefore, $\text{NACCEPTS}\varepsilon_{\text{TM}}$ is not recognizable.

11.5.3.

a) Suppose that L_1 and L_2 are recognizable languages. Let M_1 and M_2 be Turing machines that recognize L_1 and L_2, respectively. We use M_1 and M_2 to design a Turing machine M that recognizes $L_1 \cup L_2$:

   1. Let w be the input string. Run M_1 and M_2 at the same time on w (by alternating between M_1 and M_2, one step at a time).
   2. As soon as either machine accepts, accept. If they both reject, reject.

   If $w \in L_1 \cup L_2$, then either M_1 or M_2 accepts, causing M to accept. On the other hand, if M accepts w, then it must be that either M_1 or M_2 accepts, implying that $w \in L_1 \cup L_2$. Therefore, $L(M) = L_1 \cup L_2$.

b) One solution is to adapt the proof of part (a) for union. All that's needed is to change Step 2 of M:

   > If both machines accept, accept. As soon as either one rejects, reject.

Then, if $w \in L_1 \cap L_2$, then both M_1 and M_2 accept, causing M to accept. On the other hand, if M accepts w, then it must be that both M_1 and M_2 accept, implying that $w \in L_1 \cap L_2$. Therefore, $L(M) = L_1 \cap L_2$.

Another solution is to design a different, and somewhat simpler, M:

1. Let w be the input string. Run M_1 on w.
2. If M_1 rejects, reject.
3. Otherwise (if M_1 accepts), run M_2 on w.
4. If M_2 rejects, reject. Otherwise, accept.

We can show that this M is correct by using the same argument we used for the previous M.

c) Suppose that L_1 and L_2 are recognizable languages. Let M_1 and M_2 be Turing machines that recognize L_1 and L_2, respectively. We use M_1 and M_2 to design a Turing machine M that recognizes $L_1 L_2$. Here's a description of M in pseudocode:

```
Let w be the input and let n be the length of w
If w is empty, accept
For each k in {0, 1, 2, ..., n}, in parallel
   Let x be the string that consists of the
      first k symbols of w
   Let y be the string that consists of the
      remaining symbols of w
   In parallel, run M1 on x and M2 on y
   If, at any point, both machines have accepted
      Accept
Reject
```

If $w \in L_1 L_2$, then $w = xy$ with $x \in L_1$ and $y \in L_2$. This implies that for one of the partitions, M will find that both M_1 and M_2 accept and

M will accept.

It's easy to see that M accepts w only if w is of the form xy with $x \in L_1$ and $y \in L_2$.

(Note that the last step of M, the reject instruction, will be reached only if M_1 and M_2 halt on every possible partition of w.)

d) Suppose that L is a recognizable language. Let M be a TM that recognizes this language. Here's a TM M' that recognizes L^*:

```
Let w be the input and let n be the length of w
If w is empty, accept
For each k in {1, 2, ..., n}, in parallel
   For every partition of w into k substrings
         s[1], ..., s[k], in parallel
      Run M on each s[i]
      If, at any point, M has accepted all the
           substrings
         Accept
Reject
```

If $w \in L^*$, then either $w = \varepsilon$ or $w = s_1 \cdots s_k$ for some $k \geq 1$ with every $s_i \in L$. If $w = \varepsilon$, then M' accepts. If it's the other case, then in the branch of the parallel computation that corresponds to substrings $s_1, \ldots, s_k$, M' will find that M accepts every s_i and M' will accept.

On the other hand, it's easy to see that M' accepts w only if $w \in L^*$.

(Note that the last step of M', the reject instruction, will be reached only if M halts on every possible partition of w.)