Automata and Computability

Solutions to Exercises

Spring 2017

Alexis Maciel
Department of Computer Science
Clarkson University

Copyright © 2017 Alexis Maciel
# Contents

## Preface

1. Introduction

2. Finite Automata
   - 2.1 Turing Machines
   - 2.2 Introduction to Finite Automata
   - 2.3 Formal Definition
   - 2.4 More Examples
   - 2.5 Closure Properties

3. Nondeterministic Finite Automata
   - 3.1 Introduction
   - 3.2 Formal Definition
   - 3.3 Equivalence with DFA’s
   - 3.4 Closure Properties

4. Regular Expressions
   - 4.1 Introduction
   - 4.2 Formal Definition
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>More Examples</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Converting Regular Expressions into DFA’s</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Converting DFA’s into Regular Expressions</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Precise Description of the Algorithm</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Nonregular Languages</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Some Examples</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>The Pumping Lemma</td>
<td>53</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Context-Free Languages</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Introduction</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Formal Definition of CFG’s</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>More Examples</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Ambiguity and Parse Trees</td>
<td>58</td>
</tr>
</tbody>
</table>
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 solution in this document, take the time to figure out why. Did you make a mistake? Did you forget some-
thing? 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 1. INTRODUCTION
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.

Missing edges go to a garbage state.
2.2.5.
2.2.7.

```plaintext
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.

```plaintext
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 \( \Sigma \) is the set of all characters that appear on a standard keyboard and \( \delta \) 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 \( \Sigma \) is the set of all characters that appear on a standard keyboard and \( \delta \) 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 \(\Sigma\) is the set of all characters that appear on a standard keyboard and \(\delta\) is defined as follows:

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

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

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

\[
\delta(q_5, c) = q_5 \text{ for every } c
\]
2.4. More Examples

2.4.1.

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

a)

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

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: \((Q, \{0, 1\}, \delta, q_0, F)\) where

\[
Q = \{ q_w \mid w \in \{0, 1\}^* \text{ and } w \text{ has length } k \}
\]

\[
q_0 = q_{w_0} \quad \text{where } w_0 = 0^k \text{ (that is, a string of } k \text{ 0’s)}
\]

\[
F = \{ q_w \in Q \mid w \text{ starts with a } 1 \}
\]

and \( \delta \) 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)

\[
\text{Diagram for } q_0 \xrightarrow{0} q_0, \quad q_0 \xrightarrow{1} q_1, \quad q_1 \xrightarrow{0, 1} q_1
\]

b)

\[
\text{Diagram for } q_0 \xrightarrow{0} q_0, \quad q_0 \xrightarrow{1} q_1, \quad q_1 \xrightarrow{0} q_1, \quad q_1 \xrightarrow{1} q_2, \quad q_2 \xrightarrow{0, 1} q_2
\]
c) 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.

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

\[
Q = \{q_0, q_1, q_2, \ldots, q_{k-1}\}
\]

\[
\Sigma = \{0, 1, 2, \ldots, 9\}
\]

\[
F = \{q_0\}
\]

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

\[
\delta(q_i, c) = q_j \quad \text{where} \quad j = (i \cdot 10 + c) \mod k.
\]
2.4. MORE EXAMPLES

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

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

$$\Sigma = \{[abc] | a, b, c \in \{0, 1, 2, \ldots, 9\}\}$$

$$F = \{q_0\}$$

and $\delta$ is defined as follows:

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

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

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

and \(\delta\) is defined as follows:

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

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

\[
\delta(q_2, [abc]) = q_2, \quad \text{for every } [abc] \in \Sigma
\]
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)
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.
2.5. CLOSURE PROPERTIES

a)
b)
2.5. CLOSURE PROPERTIES

c)
2.5. CLOSURE PROPERTIES

d)

\begin{center}
\begin{tikzpicture}
    \node[state] (q0) at (0,0) {$q_0$};
    \node[state] (q1) at (2,0) {$q_1$};
    \node[state] (q2) at (4,0) {$q_2$};
    \draw[->] (q0) -- node[above] {0} (q1);
    \draw[->] (q1) -- node[above] {0} (q2);
    \draw[->] (q2) -- node[above] {0,1} (q0);
    \draw[->] (q0) -- node[below] {1} (q1);
    \draw[->] (q1) -- node[below] {1} (q2);
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tikzpicture}
    \node[state] (q0) at (0,0) {$q_0$};
    \node[state] (q1) at (2,0) {$q_1$};
    \draw[->] (q0) -- node[above] {1} (q1);
    \draw[->] (q1) -- node[above] {1} (q0);
    \draw[->] (q0) -- node[below] {0} (q1);
    \draw[->] (q1) -- node[below] {0} (q0);
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tikzpicture}
    \node[state] (q00) at (0,0) {$q_{00}$};
    \node[state] (q10) at (2,0) {$q_{10}$};
    \node[state] (q20) at (4,0) {$q_{20}$};
    \node[state] (q01) at (0,-2) {$q_{01}$};
    \node[state] (q11) at (2,-2) {$q_{11}$};
    \node[state] (q21) at (4,-2) {$q_{21}$};
    \draw[->] (q00) -- node[above] {1} (q10);
    \draw[->] (q10) -- node[above] {1} (q20);
    \draw[->] (q20) -- node[above] {1} (q00);
    \draw[->] (q00) -- node[below] {0} (q01);
    \draw[->] (q01) -- node[below] {0} (q00);
    \draw[->] (q00) -- node[below] {0} (q10);
    \draw[->] (q10) -- node[below] {0} (q00);
    \draw[->] (q00) -- node[below] {0} (q20);
    \draw[->] (q20) -- node[below] {0} (q00);
    \draw[->] (q01) -- node[below] {1} (q11);
    \draw[->] (q11) -- node[below] {1} (q21);
    \draw[->] (q21) -- node[below] {1} (q01);
    \draw[->] (q01) -- node[below] {0} (q11);
    \draw[->] (q11) -- node[below] {0} (q21);
    \draw[->] (q21) -- node[below] {0} (q01);
\end{tikzpicture}
\end{center}
2.5.5. In both cases, missing edges go to a garbage state.

a) 

\[ \begin{array}{c}
q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4 \\
0 \quad 1 \quad 1 \quad 1 \\
# \quad 0 \quad 0 \\
\end{array} \]

b) The dashed state and edge could be deleted.

\[ \begin{array}{c}
q_0 \rightarrow q_1 \rightarrow q_3 \rightarrow q_4 \\
0 \quad 0 \quad 1 \quad 1 \\
\end{array} \]
Chapter 3

Nondeterministic Finite Automata

3.1 Introduction

3.1.3.

a)
b) 

```
q_0 \xrightarrow{0,1} q_1 \\
q_1 \xrightarrow{0} q_0 \xrightarrow{1} q_2 \\
q_0 \xrightarrow{1} q_2 \xrightarrow{1} q_3 
```

c) 

```
q_0 \xrightarrow{0,1} q_1 \\
q_0 \xrightarrow{1} q_1 \xrightarrow{0,1} q_2 
```

d) 

```
q_0 \xrightarrow{0,1} q_1 \xrightarrow{1} q_2 \xrightarrow{0,1} \ldots \xrightarrow{0,1} q_k 
```

e) 

```
q_0 \xrightarrow{0} q_1 \\
q_0 \xrightarrow{1} q_1 
```
3.2. Formal Definition

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

\[
Q = \{q_0, q_1, q_2, q_3\} \\
F = \{q_3\}
\]

and \(\delta\) is defined by the following table

<table>
<thead>
<tr>
<th>(\delta)</th>
<th>0</th>
<th>1</th>
<th>(\epsilon)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_0)</td>
<td>(q_0)</td>
<td>(q_0, q_1)</td>
<td>(-)</td>
</tr>
<tr>
<td>(q_1)</td>
<td>(q_2)</td>
<td>(q_2)</td>
<td>(-)</td>
</tr>
<tr>
<td>(q_2)</td>
<td>(q_3)</td>
<td>(q_3)</td>
<td>(-)</td>
</tr>
<tr>
<td>(q_3)</td>
<td>(-)</td>
<td>(-)</td>
<td>(-)</td>
</tr>
</tbody>
</table>

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

\[
Q = \{q_0, q_1, q_2, q_3\} \\
F = \{q_3\} 
\]
and \( \delta \) is defined by the following table:

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>0</th>
<th>1</th>
<th>( \varepsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( q_1 )</td>
<td>( q_0 )</td>
<td>( - )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( q_2 )</td>
<td>( - )</td>
<td>( q_0 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( - )</td>
<td>( q_3 )</td>
<td>( q_1 )</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( q_3 )</td>
<td>( q_3 )</td>
<td>( - )</td>
</tr>
</tbody>
</table>

3.2.3.

The NFA accepts because the last two sequences end in the accepting state.
The NFA accepts because the last three sequences end in the accepting state.
3.3 Equivalence with DFA's

3.3.2.
3.3. EQUIVALENCE WITH DFA'S

3.3.3.

a)

<table>
<thead>
<tr>
<th>$\delta'$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_1$</td>
<td>$-$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_1, q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$q_1, q_2$</td>
<td>$q_1$</td>
<td>$q_1, q_2$</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>$\delta'$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_0, q_1$</td>
<td>$q_0, q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_3$</td>
<td>$-$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$-$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$q_1, q_0$</td>
<td>$q_0, q_1, q_3$</td>
<td>$q_0, q_2$</td>
</tr>
<tr>
<td>$q_0, q_2$</td>
<td>$q_0, q_1$</td>
<td>$q_0, q_2, q_3$</td>
</tr>
<tr>
<td>$q_0, q_1, q_3$</td>
<td>$q_0, q_1, q_3$</td>
<td>$q_0, q_2$</td>
</tr>
<tr>
<td>$q_0, q_2, q_3$</td>
<td>$q_0, q_1$</td>
<td>$q_0, q_2, q_3$</td>
</tr>
</tbody>
</table>
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)

\[
\begin{array}{c|cc}
\delta' & 0 & 1 \\
q_0 & q_0 & q_0, q_1 \\
q_1 & q_2 & q_2 \\
q_2 & - & - \\
q_0, q_1 & q_0, q_2 & q_0, q_1, q_2 \\
q_0, q_2 & q_0 & q_0, q_1 \\
q_0, q_1, q_2 & q_0, q_2 & q_0, q_1, q_2 \\
\end{array}
\]

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)

\[
\begin{array}{c|cc}
\delta' & 0 & 1 \\
q_0 & q_0 & q_1 \\
q_1 & q_1 & - \\
\end{array}
\]

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. EQUIVALENCE WITH DFA'S

#### 3.3.4.

**a)**

<table>
<thead>
<tr>
<th>$\delta'$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_1$</td>
<td>$-$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_1, q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$q_1, q_2$</td>
<td>$q_1$</td>
<td>$q_1, q_2$</td>
</tr>
</tbody>
</table>

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)**

<table>
<thead>
<tr>
<th>$\delta'$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_0, q_1, q_2$</td>
<td>$q_0, q_1, q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_3$</td>
<td>$-$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$-$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$q_0, q_1, q_2$</td>
<td>$q_0, q_1, q_2, q_3$</td>
<td>$q_0, q_1, q_2, q_3$</td>
</tr>
<tr>
<td>$q_0, q_1, q_2, q_3$</td>
<td>$q_0, q_1, q_2, q_3$</td>
<td>$q_0, q_1, q_2, q_3$</td>
</tr>
</tbody>
</table>

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

$$Q = Q_1 \cup Q_2$$
$$q_0 = q_1$$
$$F = F_2$$

and $\delta$ is defined as follows:

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

$$\delta(q, a) = \{\delta_i(q, a)\}, \text{ if } q \in Q_i \text{ and } a \in \Sigma.$$
3.4.4.

a) In the second to last paragraph of the proof, it is claimed that \( w = x_1 \cdots x_{k+1} \), with each \( x_i \in A \). It is true that \( x_1, \ldots, x_k \) are all in \( A \) because they must lead from the start state to one of the original accepting states of \( M \). But this is not true for \( x_{k+1} \): that string could lead back to the start state instead of leading to one of the original accepting states of \( M \). In that case, \( x_{k+1} \) wouldn’t be in \( A \) and we wouldn’t be able to conclude that \( w \) is in \( A^* \).

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

![DFA Diagram](image)

\[ M \]

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

![NFA Diagram](image)

\[ N \]

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. One proof is to notice that \( A^+ = AA^* \). Since the class of regular languages
is closed under star and concatenation, we also get closure under the plus operation.

An alternative proof is to modify 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 \( \epsilon \) transitions (except after \( x_k \)).

Conversely, if \( w \) is accepted by \( N \), then it must be that \( N \) uses the new \( \epsilon \) “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:

![DFA Diagram]

This DFA can be turned into an equivalent NFA \( N \) with a single accepting state as follows:
3.4. CLOSURE PROPERTIES

That is, we add a new accepting state, an \( \varepsilon \) 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 \( L^R \). 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 \( w \in L^R \). It is easy to see that the reverse is also true.
Chapter 4

Regular Expressions

4.1 Introduction

4.1.5.

a) \((- \cup \epsilon)D D^*.\)

b) \((- \cup \epsilon)D D^* \cup (- \cup \epsilon)D^* \cdot D D^*.\)

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

d) \(D^7 \cup D^{10} \cup D^3 \cdot D^4 \cup D^3 \cdot D^3 \cdot D^4.\)

4.2 Formal Definition

There are no exercises in this section.
4.3  More Examples

4.3.1. $0 \cup 1 \cup 0 \Sigma^* 0 \cup 1 \Sigma^* 1$.

4.3.2.

a) $0 \Sigma^* 1$.

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

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

4.3.3.

a) $(0 0 \cup 1 1) \Sigma^*$.

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

c) $\Sigma^* 1 \Sigma$.

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

4.3.4.

a) $\Sigma^* 1 \Sigma^*$.

b) $0^* 10^*$.

c) $\Sigma^* 1 \Sigma^* 1 \Sigma^*$.

d) $0^* \cup 0^* 10^*$.

e) $(\Sigma^* 1)^k \Sigma^*$.

4.3.5.

a) $\epsilon \cup 1 \Sigma^* \cup \Sigma^* 0$.

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

### 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)
4.4. CONVERTING REGULAR EXPRESSIONS INTO DFA'S

b)
c)

```
\begin{tikzpicture}
  \node[state] (0) {};
  \node[state] (1) at (1,0) {};
  \node[state] (2) at (2,0) {};
  \node[state] (3) at (3,0) {};
  \path[->] (0) edge [loop above] node {$\varepsilon$} (0);
  \path[->] (0) edge node {$\varepsilon$} (1);
  \path[->] (1) edge node {$0$} (2);
  \path[->] (1) edge [loop right] node {$\varepsilon$} (1);
\end{tikzpicture}
```

d)

```
\begin{tikzpicture}
  \node[state] (0) {};
  \node[state] (1) at (1,0) {};
  \node[state] (2) at (2,0) {};
  \node[state] (3) at (3,0) {};
  \node[state] (4) at (4,0) {};
  \path[->] (0) edge [loop above] node {$\varepsilon$} (0);
  \path[->] (0) edge node {$\varepsilon$} (1);
  \path[->] (1) edge node {$1$} (2);
  \path[->] (1) edge [loop right] node {$\varepsilon$} (1);
  \path[->] (2) edge node {$\varepsilon$} (3);
  \path[->] (3) edge node {$\varepsilon$} (4);
  \path[->] (3) edge [loop right] node {$\varepsilon$} (3);
\end{tikzpicture}
```
4.5 Converting DFA’s into Regular Expressions

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

\[ q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{\varepsilon} \]

We then remove state \( q_1 \):

\[ q_0 \xrightarrow{01^*1} q_2 \xrightarrow{\varepsilon} \]

We remove state \( q_2 \):

\[ 01^*1(0 \cup 01^*1)^*1 \]

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:

![Diagram showing the addition of a new accepting state](image)

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

![Diagram showing the removal of state $q_2$](image)

We remove state $q_1$:

![Diagram showing the removal of state $q_1$](image)

The final regular expression is $(0 \cup 0^+1)^*0^+$. 
4.6 Precise Description of the Algorithm

4.6.1. The GNFA is \((Q, \{a, b, c\}, \delta, q_0, F)\) where

\[
Q = \{q_0, q_1, q_2\} \\
F = \{q_2\}
\]

and \(\delta\) is defined by the following table:

<table>
<thead>
<tr>
<th>(\delta)</th>
<th>(q_0)</th>
<th>(q_1)</th>
<th>(q_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_0)</td>
<td>(a \cup b^+c)</td>
<td>(b)</td>
<td>(c \cup b^+a)</td>
</tr>
<tr>
<td>(q_1)</td>
<td>(c)</td>
<td>(b)</td>
<td>(a)</td>
</tr>
<tr>
<td>(q_2)</td>
<td>(b)</td>
<td>(c)</td>
<td>(a)</td>
</tr>
</tbody>
</table>

4.6.2. The GNFA is \((Q, \{a, b, c\}, \delta, q_0, F)\) where

\[
Q = \{q_0, q_2\} \\
F = \{q_2\}
\]

and \(\delta\) is defined by the following table:

<table>
<thead>
<tr>
<th>(\delta)</th>
<th>(q_0)</th>
<th>(q_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_0)</td>
<td>(a \cup b^+c)</td>
<td>(c \cup b^+a)</td>
</tr>
<tr>
<td>(q_2)</td>
<td>(b \cup cb^*c)</td>
<td>(a \cup cb^*a)</td>
</tr>
</tbody>
</table>
5.1.1. Suppose that \( L = \{0^n1^{2n} \mid n \geq 0\} \) is regular. Let \( M \) be a DFA that recognizes \( L \) and let \( n \) be the number of states of \( M \).

Consider the string \( w = 0^n1^{2n} \). 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:

\[
\begin{array}{c}
\overset{0^n}{\_} \overset{0^{j-i}}{\_} \overset{1^{2n}}{\_}
\end{array}
\]

This implies that the string \( 0^i0^{n-j}1^{2n} = 0^{n-(j-i)}1^{2n} \) is also accepted. But
since this string no longer has exactly \( n \) 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 \( L = \{0^i1^j \mid 0 \leq i \leq 2j\} \) is regular. Let \( M \) be a DFA that recognizes \( L \) and let \( n \) be the number of states of \( M \).

Consider the string \( w = 0^{2n}1^n \). As \( M \) reads the first \( n \) 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:

![Diagram showing computation](image)

Now consider going twice around the loop. This implies that the string \( 0^i0^{2(j-i)}0^{2n-j}1^n = 0^{2n+(j-i)}1^n \) is also accepted. But since this string has more than \( 2n \) 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 \( L = \{ww^R \mid w \in \{0, 1\}^*\} \) is regular. Let \( M \) be a DFA that recognizes \( L \) and let \( n \) be the number of states of \( M \).

Consider the string \( w = 0^n110^n \). As \( M \) reads the first \( n \) 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 $0^i0^{n-j}110^n = 0^{n-(j-i)}110^n$ 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 $L = \{0^i1^j \mid i \leq j\}$. Suppose that $L$ is regular. Let $p$ be the pumping length. Consider the string $w = 0^p1^p$. 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. $y \neq \epsilon$.
3. $xy^kz \in L$, for every $k \geq 0$.

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

**5.2.2.** Let $L = \{1^i \#1^j \#1^{i+j}\}$. Suppose that $L$ is regular. Let $p$ be the pumping length. Consider the string $w = 1^p \#1^p \#1^{2p}$. 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 $w = 1^p \# 2^p \# 3^p$. 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 $k \geq 0$.

Since $|xy| \leq p$, we have that $y$ contains only 1’s from the first part of the string. Therefore, $xy^2z = 1^{p+|y|} \# 1^p \# 1^{2p}$. In other words, since $|y| \geq 1$, the first number in this string was changed but not the other two, making impossible for the sum of the first two numbers to equal the third. Therefore, $xy^2z$ is not in $L$. 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.
6.1.6.

a) 

\[ I \rightarrow SN \]
\[ S \rightarrow - | \varepsilon \]
\[ N \rightarrow DN | D \]
\[ D \rightarrow 0 | \cdots | 9 \]
b) 

\[ \begin{align*}
R & \rightarrow SN_1 \mid SN_0 \cdot N_1 \\
S & \rightarrow \epsilon \\
N_0 & \rightarrow DN_0 \mid \epsilon \\
N_1 & \rightarrow DN_0 \\
D & \rightarrow 0 \mid \cdots \mid 9
\end{align*} \]

c) 

\[ \begin{align*}
I & \rightarrow _R R_1 \mid LR_0 \\
R_0 & \rightarrow _R R_0 \mid LR_0 \mid DR_0 \mid \epsilon \\
R_1 & \rightarrow R_0 LR_0 \mid R_0 DR_0 \\
L & \rightarrow a \mid \cdots \mid z \mid A \mid \cdots \mid Z \\
D & \rightarrow 0 \mid \cdots \mid 9
\end{align*} \]

6.2 Formal Definition of CFG's

There are no exercises in this section.
6.3. **MORE EXAMPLES**

6.3 More Examples

6.3.1.

a) 

\[ S \rightarrow 0S0 \mid 1 \]

b) 

\[ S \rightarrow 0S0 \mid 1S1 \mid \varepsilon \]

c) 

\[ S \rightarrow 0S11 \mid \varepsilon \]

d) Here’s one solution:

\[ S \rightarrow ZS1 \mid \varepsilon \]

\[ Z \rightarrow 0 \mid \varepsilon \]

Here’s another one:

\[ S \rightarrow 0S1 \mid T \]

\[ T \rightarrow T1 \mid \varepsilon \]
6.3.2.

\[ S \rightarrow 1S1 | \#T \]
\[ T \rightarrow 1T1 | \# \]

6.3.3.

\[ S \rightarrow (S)S | [S]S | \{S\}S | \epsilon \]

6.3.4. A string of properly nested parentheses is either \( () \) or a string of the form \( (u)v \) where \( u \) and \( v \) are either empty or strings of properly nested parentheses. Here's a grammar that paraphrases this definition:

\[ S \rightarrow (U)U | () \]
\[ U \rightarrow S | \epsilon \]

Here's an alternative that essentially incorporates the rules for \( U \) into the rules for \( S \):

\[ S \rightarrow (S)S | ()S | (S) | () \]

6.4 Ambiguity and Parse Trees

6.4.4. Two parse trees in the first grammar:
6.4. AMBIGUITY AND PARSE TREES

\[
\begin{array}{c}
E \\
E + E \\
E * E \\
a \\
a \\
\end{array}
\quad
\begin{array}{c}
E \\
E * E \\
a \\
E + E \\
a \\
\end{array}
\]
The unique parse tree in the second grammar: