# PART II

## Finite and Regular Languages

# Part II
# Finite and Regular Languages

## 6 Finite Languages

We'll start with the simplest class of languages: the *Finite Languages*—finite sets of strings over some alphabet. While this, in itself, is a perfectly precise definition of the class, it will be useful to characterize it inductively as well.

**Definition 18 (Finite Languages)** *For any alphabet $\Sigma$:*

- *$\emptyset$ is a finite language over $\Sigma$,*
- *The singleton set $\{\varepsilon\}$ is a finite language over $\Sigma$,*
- *For each $\sigma \in \Sigma$, the singleton set $\{\sigma\}$ is a finite language over $\Sigma$,*
- *If $L_1$ and $L_2$ are finite languages over $\Sigma$ then:*
  - *$L_1 \cdot L_2$ is a finite language over $\Sigma$,*
  - *$L_1 \cup L_2$ is a finite language over $\Sigma$.*
- *Nothing else is a finite language over $\Sigma$.*

Which is to say, the finite languages are exactly those that can be constructed using concatenation and union from the empty language and the languages consisting of just the empty string or the unit strings of $\Sigma$.

It remains to verify that this actually defines the class of all finite languages. Typically, it is easiest to do this in two steps: show that every language constructed as in the definition is finite (the class contains *only* finite languages) and show that every finite language can be so constructed (it contains *all* the finite languages).

**Lemma 1** *Any language in the class of Definition 18 is finite.*

**Proof:** As the class is defined inductively, we will prove this by structural induction (i.e., induction on the construction of the language).

(Basis:)

Clearly $\emptyset$, $\varepsilon$, and all the singleton languages consisting of just a unit string from $\Sigma$ are finite.

(IH:)

Suppose that $L$ is obtained from $L_1$ and $L_2$, both in the class of Definition 18, by either concatenation or union. Assume, for induction, that $L_1$ and $L_2$ are finite.

(Ind:)

To show that $L$ must also be finite: The strings in $L_1 \cdot L_2$ are all strings obtained by concatenating a string from $L_1$ and a string from $L_2$. Thus,

$$\mathbf{card}(L_1 \cdot L_2) \leq \mathbf{card}(L_1 \times L_2) = \mathbf{card}(L_1) \cdot \mathbf{card}(L_2).$$

9. Why is it $\leq$ and not just $=$?

The strings in $L_1 \cup L_2$ are all strings in either $L_1$ or $L_2$. Thus,

$$\mathbf{card}(L_1 \cup L_2) \leq \mathbf{card}(L_1) + \mathbf{card}(L_2).$$

10. Again, why is it $\leq$ and not $=$?

Since both the product and the sum of finite numbers are finite, $L$ is also of finite. $\dashv$

We now turn to establishing that every language over $\Sigma$ that contains finitely many strings is constructible as in Definition 18. We'll do this in two steps: first we will establish that every singleton language over $\Sigma$ is constructible, then we will use that result to establish that every finite language over $\Sigma$ is constructible. (At this point it you should have a pretty good idea how we are going to proceed.)

**Lemma 2** *Every singleton language over $\Sigma$ is constructible as in Definition 18.*

**Proof:** Suppose $L$ is a singleton language over $\Sigma$. Then $L$ consists of just a single string; let $w$ denote that string. We will proceed by induction on the length of $w$. (Note that this is the same as induction on the structure of $w$ given the inductive definition of strings of Definition 1.)

(Basis:)

If $w = \varepsilon$ then $w$ is constructed by one of the base cases of the definition.

(IH:)

Suppose that $w = v\sigma$ for some $v \in \Sigma^*$ and $\sigma \in \Sigma$. Assume, for induction that $v$ is constructible as in Definition 18.

(Ind:)

The language $\{\sigma\}$ is constructible by one of the base cases of the definition. $L$ is then constructible as $\{v\} \cdot \{\sigma\}$. ⊣

**Lemma 3** *Every finite language over $\Sigma$ is constructible as in Definition 18.*

**Proof:** Suppose $L$ is a finite language. We proceed by induction on the cardinality of $L$.

(Basis:)

Suppose $\mathbf{card}(L) = 0$. Then $L = \emptyset$ and is constructible by a base case of the definition.

(IH:)

Suppose all sets of cardinality $n$ are constructible and that $\mathbf{card}(L) = n+1$.

(Ind:)

Let $w$ be any string in $L$. Then $\{w\}$ is constructible by Lemma 2 and $L \setminus \{w\}$ has cardinality $n$ and hence, by IH, is constructible. Since $L$ is the union of these two constructible languages, it is constructible as well. ⊣

11. Why does this proof not work for infinite languages as well?
[**Hint:** Why does the induction fail?]

Putting these together, we get:

**Claim 3** *The class of languages defined in Definition 18 includes all and only the finite languages.*

12. Recall that we require alphabets to be finite. What happens to this definition if $\Sigma$ is infinite—does it still define the class of finite languages?
[**Hint:** Surely the fact that $\Sigma$ is infinite does not make the class of languages that can be constructed any smaller: all finite languages over $\Sigma$ will still be constructible. The question hinges only on Lemma 1, which claims that *only* finite languages are constructible. Does the proof of Lemma 1 depend on the finiteness of $\Sigma$?]

# 7 Regular Languages and Regular Expressions

The *Regular Languages* are those obtainable by extending our descriptive mechanism with the Kleene closure, in some sense the simplest means of defining infinite languages.

**Definition 19 (Regular Languages (Kleene))** *The* regular languages *are those obtainable from the finite languages by union, concatenation, or Kleene closure.*

Since the finite languages are those obtainable from the empty language and the singleton languages consisting of just the empty string or a unit string we can define the regular languages inductively simply by adding the Kleene closure to Definition 18.

**Definition 20 (Regular Languages)** *For any alphabet $\Sigma$:*

- $\emptyset$ *is a regular language over* $\Sigma$,
- *The singleton set* $\{\varepsilon\}$ *is a regular language over* $\Sigma$,
- *For each* $\sigma \in \Sigma$, *the singleton set* $\{\sigma\}$ *is a regular language over* $\Sigma$,
- *If* $L_1$ *and* $L_2$ *are regular languages over* $\Sigma$ *then:*
  - $L_1 \cdot L_2$ *is a regular language over* $\Sigma$,
  - $L_1 \cup L_2$ *is a regular language over* $\Sigma$.
  - ${L_1}^*$ *is a regular language over* $\Sigma$.
- *Nothing else is a regular language over* $\Sigma$.

To facilitate reasoning about the regular languages, Kleene introduced the algebra of *Regular Expressions.*

**Definition 21 (Regular Expressions)** *For any alphabet $\Sigma$:*

- $\emptyset$ *is a regular expression over* $\Sigma$,
- $\varepsilon$ *is a regular expression over* $\Sigma$,
- *For each* $\sigma \in \Sigma$, $\sigma$ *is a regular expression over* $\Sigma$,

- *If $R$ and $S$ are regular expressions over $\Sigma$ then:*
  - *$(R \cdot S)$ is a regular expression over $\Sigma$,*
  - *$(R + S)$ is a regular expression over $\Sigma$,*
  - *$(R^*)$ is a regular expression over $\Sigma$.*
- *Nothing else is a regular expression over $\Sigma$.*

Note that, by this definition, all regular expressions must be fully parenthesized. This is generally relaxed, adopting precedence for the operators with '*' binding most tightly, followed by '$\cdot$' and then '+'. Also, '$\cdot$' is usually dropped, with the concatenation of two expressions being indicated simply by their juxtaposition.

Each regular expression stands for a particular regular language, its *denotation*. We will use the notation $L(R)$ for the language denoted by $R$.

**Definition 22 (Denotation of a regular expression)** *For any regular expression $R$ over an alphabet $\Sigma$:*

$$L(R) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } R = \emptyset, \\ \{\varepsilon\} & \text{if } R = \varepsilon, \\ \{\sigma\} & \text{if } R = \sigma \in \Sigma, \\ L(S_1) \cdot L(S_2) & \text{if } R = (S_1 \cdot S_2), \\ L(S_1) \cup L(S_2) & \text{if } R = (S_1 + S_2), \\ L(S)^* & \text{if } R = (S^*). \end{cases}$$

It should be clear that a language is regular iff it is the denotation of a regular expression. (One can prove this by induction on the structure of the set—for the only if direction—and induction on the structure of the expression—for the if direction.)

You should verify for yourself that the denotation is well defined: that each regular expression denotes exactly one language. This follows from the fact that the definition of the denotation includes a case for each case of the definition of the class of expressions and that each of the set building operations '$\cdot$', '$\cup$', and '*' are well defined.

13. What about the converse of this? Is it possible for a given language to be the denotation of more than one regular expression? If so, give a simple example.
    [**Hint:** This is, in essence, asking if there are any regular languages that can be constructed in more than one way.]

14. Show that the basis expression '$\varepsilon$' is redundant; every set that can be defined using it can also be defined without it.
[**Hint:** Find a regular expression in which '$\varepsilon$' does not occur but which denotes $\{\varepsilon\}$.]

It should be emphasized that there is never any question about the meaning of a regular expression—its meaning is exactly as defined in Definition 22. One simply carries out the definition recursively.

**Example:** What is the denotation of '$(b^*a + b)^*$'?
Proceeding in excruciating detail:

$$\begin{aligned} L((b^*a + b)^*) &= (L(b^*a + b))^* \\ &= (L(b^*a) \cup L(b))^* \\ &= ((L(b^*) \cdot L(a)) \cup L(b))^* \\ &= (((L(b))^* \cdot \{a\}) \cup \{b\})^* \\ &= ((((\{b\})^* \cdot \{a\}) \cup \{b\})^* \end{aligned}$$

You may wish to simplify somewhat you are not required to (and be careful if you do—it's easy to corrupt an otherwise correct answer).

15. What is the denotation of '$(b^*((ab^*)^*a + \varepsilon))^*$'? Write out the each step of the translation following Definition 22.

## 7.1 The Algebra of Regular Expressions

The idea that there may be many distinct expressions with the same meaning should be familiar from the algebra of numbers. Just as we say that two algebraic expressions are equal if the denote the same number, we will say that two regular expressions are equivalent iff they denote the same set. And, just as we can establish such equivalences in the algebra of numbers by applying the familiar laws of addition, multiplication, etc., we can establish equivalences between regular expressions by applying algebraic properties of the regular operations on sets:

**Definition 23 (Properties of the Regular Operations)** *If $R$ and $S$ are regular expressions (over any alphabet), we will say*

$$R = S \stackrel{\text{def}}{\Longleftrightarrow} L(R) = L(S).$$

*For all regular expressions $R$, $S$, and $T$:*

| | | |
|---|---|---|
| R1) | $R + S = S + R$ | *(+ commutative)* |
| R2) | $R + \emptyset = \emptyset + R = R$ | *($\emptyset$ is unit for +)* |
| R3) | $R + R = R$ | *(idempotency of +)* |
| R4) | $(R + S) + T = R + (S + T)$ | *(+ associative)* |
| R5) | $R\emptyset = \emptyset R = \emptyset$ | *($\emptyset$ is zero for $\cdot$)* |
| R6) | $R\varepsilon = \varepsilon R = R$ | *($\varepsilon$ is unit for $\cdot$)* |
| R7) | $(RS)T = R(ST)$ | *($\cdot$ associative)* |
| R8) | $R(S + T) = RS + RT$ | |
| R9) | $(R + S)T = RT + ST$ | *($\cdot$ distributes over +)* |
| R10) | $(\emptyset)^* = \varepsilon$. | |
| R11) | $R^* = (R + \varepsilon)^*$. | |
| R12) | $R^* = R^*R + \varepsilon$. | |

Each of these laws (or axioms) is justified by the properties of the corresponding operations on sets:

| | | |
|---|---|---|
| R1) | $R + S = S + R$ | since $L(R) \cup L(S) = L(S) \cup L(R)$ |
| R2) | $R + \emptyset = \emptyset + R = R$ | since $L(R) \cup \emptyset = \emptyset \cup L(R) = L(R)$ |
| | $\vdots$ | |
| R10) | $(\emptyset)^* = \varepsilon$ | since $L((\emptyset)^*) = \bigcup_{i \geq 0}[\emptyset^i] = \emptyset^0 \cup \bigcup_{i \geq 1}[\emptyset^i]$ $= \{\varepsilon\} \cup \bigcup_{i \geq 1}[\emptyset] = \{\varepsilon\} \cup \emptyset = \{\varepsilon\} = L(\varepsilon)$ |
| | $\vdots$ | |

These axioms, along with uniform substitution of expressions for variables (i.e., replacing each $R$ in an equation with the same expression) and an inference rule that says, "if $P = PQ + R$ and $Q + \varepsilon \neq Q$ then $P = RQ^*$" derive all true equations in the algebra of regular expressions. (We will see more of this last inference rule shortly.) They are somewhat easier to apply with the help of some additional identities (which are, of course, redundant in that they are implied by the axioms):

| | |
|---|---|
| I1 | $R^* = R^*R^* = (R^*)^* = R + R^*$ |
| I2 | $R^* = \varepsilon + R + R^2 + R^3 + \cdots + R^kR^*$, for any $k \geq 0$ |
| I3 | $R^*R = RR^*$ |
| I4 | $(R + S)^* = (R^* + S^*)^* = (R^*S^*)^* = (R^*S)^*R^*$ |
| I5 | $R(SR)^* = (RS)^*R$ |

**Example:** To show that the regular expressions of Exercise 7 and Example 7

denote the same language:

$$
\begin{array}{lll}
(b^*((ab^*)^*a+\varepsilon))^* & & \\
\quad = & (b^*(ab^*)^*a+b^*\varepsilon)^* & \text{R8 } (R=b^*,\ S=(ab^*)^*a,\ T=\varepsilon) \\
\quad = & (b^*(ab^*)^*a+b^*)^* & \text{R6 } (R=b^*) \\
\quad = & (b^*a(b^*a)^*+b^*)^* & \text{I5 } (R=a,\ S=b^*) \\
\quad = & ((b^*a)^*b^*a+b^*)^* & \text{I3 } (R=b^*a) \\
\quad = & ((b^*a)^*b^*a+b^*b+\varepsilon)^* & \text{R12 } (R=b) \\
\quad = & ((b^*a)^*b^*a+b^*b+\varepsilon+\varepsilon)^* & \text{R3 } (R=\varepsilon) \\
\quad = & ((b^*a)^*b^*a+\varepsilon+b^*b+\varepsilon)^* & \text{R1 } (R=b^*b,\ S=\varepsilon) \\
\quad = & ((b^*a)^*+b^*)^* & \text{R12 (twice) } (R_1=b^*a,\ R_2=b) \\
\quad = & (b^*a+b)^* & \text{I4 } (R=b^*a,\ S=b)
\end{array}
$$

In practice it is almost always easier to appeal to the properties of the denotations of the regular expressions, particularly when establishing identities.

**Example:** To show that $(R^*)^* = R^*$ it suffices to show that $(L^*)^* = L^*$ for all languages $L$. We can establish this by showing inclusion each way:
To show that $L^* \subseteq (L^*)^*$: Suppose $w \in L^*$. Then $w \in \bigcup_{i\geq 0}[L^i]$ and, in particular, $w \in L^k$ for some $k \geq 0$. Then

$$w \in L^k = (L^k)^1 \subseteq \bigcup_{j\geq 0}[(\bigcup_{i\geq 0}[L^i])^j] = (L^*)^*.$$

To show that $(L^*)^* \subseteq L^*$: Suppose $w \in (L^*)^*$. Then $w \in \bigcup_{j\geq 0}[(\bigcup_{i\geq 0}[L^i])^j]$ and, in particular, $w \in (L^k)^l$ for some $k, l \geq 0$. Then

$$w \in (L^k)^l = L^{kl} \subseteq \bigcup_{i\geq 0}[L^i] = L^*.$$

**Example:** To show that $R(SR)^* = (RS)^*R$ it suffices to show that $L(ML)^* = (LM)^*L$, which we do, again, by showing inclusion both ways.
To show that $L(ML)^* \subseteq (LM)^*L$: Suppose that $w \in L(ML)^*$. Then $w \in L \cdot \bigcup_{i\geq 0}[(ML)^i]$ and, in particular, $w \in L(ML)^k$ for some $k \geq 0$. We proceed by induction on $k$.

(Basis)

If $k = 0$ then

$$L(ML)^0 = L\{\varepsilon\} = L = \{\varepsilon\}L = (LM)^0L.$$

(IH)
Suppose $k > 0$ and $L(ML)^{k-1} = (LM)^{k-1}L$.

(Ind:)
Then,

$$L(ML)^k = LML(ML)^{k-1} = LM(LM)^{k-1}L = (LM)^kL.$$

Hence, $L(ML)^k = (LM)^kL$ for all $k \geq 0$ and

$$w \in L(ML)^k = (LM)^kL \subseteq \bigcup_{i \geq 0}[(LM)^i] \cdot L = (LM)^*L.$$

The other direction is similar.

16. Show that $R^* = R + R^*$.

17. Show that $(R + S)^* = (R^*S^*)^*$.

18. Show that $R^* = R^* + \varepsilon$
[**Hint:** This one is easier using the axioms and identities. Look at the last example again.]

## 7.2 Defining Languages with Regular Expressions

The characteristic operation of regular languages is iteration. When attempting to capture a language with a regular expression one good way to start is to look for a way to split strings in the language up into blocks that repeat. This will not always be enough, but it is usually a good start.

**Example:** Give a regular expression for the set of strings over $\{a, b\}$ in which every pair of adjacent '$a$'s appears before any pair of adjacent '$b$'s. Prove that the the language denoted by your regular expression is exactly the language described.

Let's call this language $L_1$. We can start by noting that strings in this language will always have two parts (either or both of which may be empty): one in which no '$bb$' occurs followed by one in which no '$aa$' occurs. Thus, it is the concatenation of $L_{\overline{bb}}$ (the language over $\{a, b\}$ in which no '$bb$' occurs) and $L_{\overline{aa}}$ (the similar language for '$aa$'). To be complete, we should prove this assertion.

**Lemma 4** $L_1 = L_{\overline{bb}} \cdot L_{\overline{aa}}$.

**Proof:**

($L_{\overline{bb}} L_{\overline{aa}} \subseteq L$:)

Let $w \in L_{\overline{bb}} L_{\overline{aa}}$. Then $w = w_1 w_2$ where $w_1 \in L_{\overline{bb}}$ and $w_2 \in L_{\overline{aa}}$. If any $bb$ occurs in $w$ it must occur in $w_2$ and, similarly, any $aa$ must occur in $w_1$. Thus every such $aa$ must precede any such $bb$.

($L \subseteq L_{\overline{bb}} L_{\overline{aa}}$:)

Suppose $w \in L$. If no $aa$ occurs in $w$ then $w \in \{\varepsilon\} \cdot L_{\overline{aa}}$ which is a subset of $L_{\overline{bb}} L_{\overline{aa}}$. Similarly for the case in which no $bb$ occurs in $w$. Suppose, then, that at least one $aa$ and one $bb$ occur in $w$. Let $w = w_1 aa w_2$ where no $aa$ occurs in $w_2$ (i.e., the $aa$ is the last $aa$ in $w$). Then $w_2 \in L_{\overline{aa}}$ and, since any $bb$ must follow the $aa$, $w_1 aa \in L_{\overline{bb}}$. ⊣

We can now develop a regular expressions for $L_{\overline{bb}}$ and $L_{\overline{aa}}$. We'll do $L_{\overline{bb}}$; the expression for $L_{\overline{aa}}$ is, of course, similar.

The insight here is that '$b$'s only ever occur singly. Any string in the language, then, can be broken up into segments as follows:

$$\underbrace{a \cdots a}_{\geq 0} \; \overbrace{b \underbrace{a \cdots a}_{\geq 0}}^{\geq 1} \; \underbrace{b}_{\leq 1}.$$

As a regular expression: $r_{\overline{bb}} = a^*(baa^*)^*(\varepsilon + b)$.

(Note that this is not the simplest expression that denotes the language, but it is one that will facilitate the proof of its correctness.)

**Claim 4** $L_{\overline{bb}} = L(r_{\overline{bb}})$.

**Proof:**

($L(r_{\overline{bb}}) \subseteq L_{\overline{bb}}$:)

If $w \in L(r_{\overline{bb}})$ then $w = xyz$ where $x \in L(a^*)$, $y \in L((baa^*)^*)$ and $z \in L(\varepsilon + b)$. Then no '$b$'s at all occur in $x$ and at most a single '$b$' occurs in $z$. To show that no '$bb$' occurs in any string in $y$ we'll prove, by induction on the number of iterations of $(baa^*)$, the strengthened hypothesis: no '$bb$' occurs in any string in $L((baa^*)^*)$ and no string in $L((baa^*)^*)$ ends in '$b$'. This is trivially true for $\varepsilon$, the base case. For the inductive step we note that if the claim is true for all $x_1 \in L((baa^*)^n)$ then it is also true for

$$x_1 x_2 \in L((baa^*)^n) \cdot L(baa^*) = L((baa^*)^{n+1})$$

for every $x_2 \in L(baa^*)$, since the '$b$' in $x_2$ is neither preceded or followed by a '$b$' and $x_2$ ends in '$a$'. Finally we note that, if no '$bb$' occurs in any of $x$, $y$, or $z$ and neither $x$ nor $y$ ends in '$b$', then no '$bb$' occurs in their concatenation either.

($L_{\overline{bb}} \subseteq L(r_{\overline{bb}})$:)

Let $w \in L_{\overline{bb}}$. Split $w$ into substrings immediately before each '$b$'. Then

$$w = w_0 \cdot w_1 \cdot \cdots \cdot w_k,$$

for some $k \geq 0$, where no '$b$' occurs in $w_0$ and each of the $w_i$, $1 \leq i \leq k$ starts with '$b$' and contains no other '$b$'. Since $w$ contains no '$bb$', each of the $w_i$, $1 \leq i < k$ must contain at least one '$a$'. Moreover $w_k$ is either a single $b$ or is a $b$ followed by one or more '$a$'s. One of two cases holds, then; either

$$w_0 \in L(a^*),\ w_i \in L(baa^*),\ 1 \leq i < k, \quad \text{and } w_k = b,$$

or

$$w_0 \in L(a^*),\ w_i \in L(baa^*),\ 1 \leq i \leq k.$$

In either case $w \in L(r_{\overline{bb}})$. ⊣

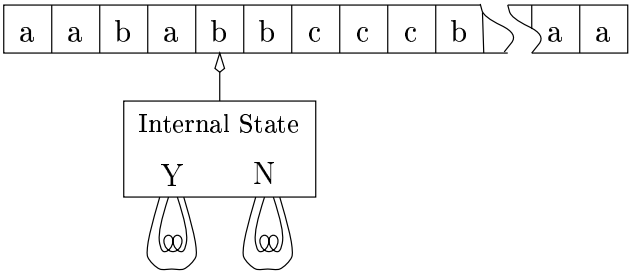
Finally, we put these together and get

$$L_1 = L(r_{\overline{bb}} \cdot r_{\overline{aa}}) = L(a^*(baa^*)^*(\varepsilon + b)b^*(abb^*)^*(\varepsilon + a)).$$

19. Write a regular expression for the language over $\{a, b\}$ in which no string contains the sequence '$bab$' as a substring. Prove that the regular expression denotes exactly that language.

# 8 Deterministic Finite-State Automata (DFAs)

The most restricted model of computation we will consider is very nearly the simplest possible model. We will assume that there is a finite bound on the amount of information the machine can store. We can model this in abstract terms by thinking of the *internal state* of the machine as being a representation of all the information it has stored. Since there is a finite bound on the amount it can store, the machine will have but finitely many states. Schematically, such a machine looks something like this:




Here the input is on a read-only tape which is scanned left to right by the machine. Each time the machine reads a symbol of the input it updates its internal state and moves the head to the right. It halts when it moves off the right end of the tape. We can fully specify the behavior of the machine by specifying its initial state, how it passes from one state to the next in response to the input, and in which states it should light the '$Y$' lamp. The machine accepts the input, of course, iff it halts with the '$Y$' lamp on.

**Definition 24** *A* Deterministic Finite-state Automaton (DFA) *is a 5-tuple:* $\langle Q, \Sigma, \delta, q_0, F\rangle$*, where:*

| | |
|---|---|
| $Q$ | *is the set of* states, |
| $\Sigma$ | *is the* input alphabet, |
| $\delta : Q \times \Sigma \to Q$ | *is the* transition function *(mapping a state and an input symbol to the next state),* |
| $q_0 \in Q$ | *is the* start *(or* initial*)* state, |
| $F \subseteq Q$ | *is the set of* final states *(or* accepting states*).* |

Note that the set $Q$ can be anything we like. It will often be useful to take it to be a set of names with the name of a state being chosen to indicate the significance of that state in the computation. The transition function of a DFA will always be *total*, i.e., defined for every $q \in Q$ and $\sigma \in \Sigma$. Thus the DFA never crashes—it will always have a next state to go to so long as there is more input to read. The only way for the DFA to halt is for it to reach the end of the tape. Since this allows the state set and input alphabet to be inferred from $\delta$, it will generally suffice to specify only $\delta$, the start state and the set of final states.

## 8.1 Computations of DFAs

In order to carry out formal proofs about the computations of such a machine we will need a precise definition of what these computations are. The way we

will approach this is identify a computation of the DFA with a formalized representation of a trace of that computation: a sequence of tuples in which each tuple represents the status of the machine one step of the computation. To fully characterize the status of the DFA at any given point in the computation we need to specify the input, the position of the read head and state of the machine at that point. Since the read head moves only towards the right, the input that has already been scanned can play no further role in the computation. Thus, we can represent both the input and the position of the read head within it using a string representing the portion of the input that remains to be read. We will refer such representations as *Instantaneous Descriptions.*

**Definition 25 (Instantaneous Description of a DFA)** *An* instantaneous description *of a DFA* $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ *is a pair* $\langle q, w\rangle \in Q \times \Sigma^*$*, where* $q$ *the current state and* $w$ *is the portion of the input under and to the right of the read head.*

The symbol being currently being read by the DFA is the first symbol of $w$. If $w$ is empty then the entire input has been scanned and the DFA has halted.

**Definition 26 (Directly Computes Relation for DFAs)**

$$\langle q, w\rangle \vdash_{\mathcal{A}} \langle p, v\rangle \stackrel{\text{def}}{\iff} w = \sigma v \text{ and } p = \delta(q, \sigma).$$

Note that this implies that $\langle q, \varepsilon\rangle$ has no successor for any $q$. IDs in which $w = \varepsilon$ are *terminal* (or *halted*) IDs; they represent the fact that the DFA has halted in state $q$. Note also that, because $\delta$ is a total function, every ID in which $w \neq \varepsilon$ has a successor and that successor is unique. (Thus, $\vdash_{\mathcal{A}}$ is *partial functional.*)

**Definition 27 (Computation of a DFA)** *A computation of a DFA* $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ *from state* $q_1$ *on input* $w_1$ *is a sequence of IDs* $\langle \langle q_1, w_1\rangle, \ldots\rangle$ *in which, for all* $i > 0$, $\langle q_{i-1}, w_{i-1}\rangle \vdash_{\mathcal{A}} \langle q_i, w_i\rangle$ *and which is closed under* $\vdash_{\mathcal{A}}$ *: for all* $i$*, if* $w_i \neq \varepsilon$ *and* $\langle q_i, w_i\rangle \vdash_{\mathcal{A}} \langle q_{i+1}, w_{i+1}\rangle$ *then* $\langle q_{i+1}, w_{i+1}\rangle$ *is included in the sequence.*

Since $\vdash_{\mathcal{A}}$ is partial functional there is exactly one computation of $\mathcal{A}$ from each ID in $Q \times \Sigma$. Since each step of a computation of a DFA consumes exactly one symbol of the input, the length of the computation from $\langle q_1, w_1\rangle$ will be $|w_1|$.

Moreover, $w_{|w_1|}$ will be $\varepsilon$. Thus, all computations of a DFA halt and they all take exactly $|w_1|$ steps.

The '$\vdash_{\mathcal{A}}$' relation captures single steps of the computations of $\mathcal{A}$. The relation $\vdash^*_{\mathcal{A}}$ holds between two IDs iff the second can be reached from the first in zero or more steps:

**Definition 28 (Computes Relation for DFAs)**
$\langle q, w\rangle \vdash^*_{\mathcal{A}} \langle p, v\rangle$ ($\langle q, w\rangle$ computes $\langle p, v\rangle$ in $\mathcal{A}$) *iff* $\langle p, v\rangle$ *occurs in the computation of* $\mathcal{A}$ *on* $\langle q, w\rangle$.

$\langle q, w\rangle \vdash^n_{\mathcal{A}} \langle p, v\rangle$ ($\langle q, w\rangle$ computes $\langle p, v\rangle$ in $n$ steps in $\mathcal{A}$) *iff* $\langle p, v\rangle$ *is the* $(n+1)^{\text{st}}$ *element of the computation of* $\mathcal{A}$ *on* $\langle q, w\rangle$. *(I.e., iff* $\langle p, v\rangle$ *is the* $n^{\text{th}}$ *successor of* $\langle q, w\rangle$*.)*

Then $\langle q, w\rangle \vdash^0_{\mathcal{A}} \langle p, v\rangle$ iff $\langle q, w\rangle = \langle p, v\rangle$ and $\langle q, w\rangle \vdash^*_{\mathcal{A}} \langle p, v\rangle$ iff $\langle q, w\rangle \vdash^n_{\mathcal{A}} \langle p, v\rangle$ for some $n$.

**Lemma 5** $\vdash^*_{\mathcal{A}}$ *is the reflexive, transitive closure of* $\vdash_{\mathcal{A}}$ .

**Proof:** ($\vdash^*$ extends $\vdash$ :)

$\langle q, w\rangle \vdash^1_{\mathcal{A}} \langle p, v\rangle$ iff $\langle q, w\rangle \vdash_{\mathcal{A}} \langle p, u\rangle$.

($\vdash^*$ is reflexive:)

$\langle q, w\rangle \vdash^0_{\mathcal{A}} \langle q, w\rangle$.

($\vdash^*$ is transitive:)

If $\langle p, u\rangle$ occurs in the computation of $\mathcal{A}$ on $\langle q, w\rangle$ then the computation of $\mathcal{A}$ on $\langle p, u\rangle$ is a suffix of the computation of $\mathcal{A}$ on $\langle q, w\rangle$. Thus, $\langle q, w\rangle \vdash^*_{\mathcal{A}} \langle p, v\rangle$ and $\langle p, v\rangle \vdash^*_{\mathcal{A}} \langle o, u\rangle$ iff $\langle o, u\rangle$ occurs in the computation of $\mathcal{A}$ on $\langle q, w\rangle$, i.e., iff $\langle q, w\rangle \vdash^*_{\mathcal{A}} \langle o, u\rangle$. ⊣

The *language accepted by a DFA* $\mathcal{A}$ (which we will denote $L(\mathcal{A})$) is the set of strings $w$ for which $\mathcal{A}$, when run from the start state with $w$ on the tape, halts in an accepting state.

**Definition 29 (Language Accepted by a DFA)** *The* language accepted by a DFA $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ *is*

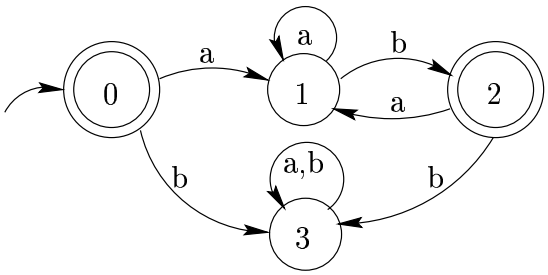
$$L(\mathcal{A}) = \{w \mid \langle q_0, w \rangle \stackrel{*}{\vdash_{\mathcal{A}}} \langle q, \varepsilon \rangle, q \in F\}$$

**Definition 30 (Recognizable Language)** *A language is* recognizable *iff it is accepted by some DFA.*

## 8.2 Transition Graphs

It turns out to generally be simpler to prove things about a DFA if we think of it as a labeled, directed graph (know as its *transition graph*):[1] $Q$ is the set of nodes with $F$ being a distinguished subset (conventionally indicated by circling them), the transition function determines the edge relation with the edge between two states being labeled with the input that causes the transition, and the initial state is indicated by an in-edge with no source. For example:

$$\begin{array}{rcl} Q & = & \{0,1,2,3\} \\ \Sigma & = & \{a,b\} \\ \delta & = & \{\langle 0,a\rangle \mapsto 1,\ \langle 0,b\rangle \mapsto 3, \\ & & \ \langle 1,a\rangle \mapsto 1,\ \langle 1,b\rangle \mapsto 2, \\ & & \ \langle 2,a\rangle \mapsto 1,\ \langle 2,b\rangle \mapsto 3, \\ & & \ \langle 3,a\rangle \mapsto 3,\ \langle 3,b\rangle \mapsto 3\ \} \\ q_0 & = & 0 \\ F & = & \{0,2\}. \end{array}$$




In these terms, as the automaton scans an input string it traces out a path in the graph, starting with $q_0$, in which the labels of the edges form the same string. Note that the requirement that $\delta$ be total corresponds to a requirement that every node in the graph has an out-edge for each symbol in $\Sigma$. It follows that every string over $\Sigma$ labels some path from $q_0$. A string is accepted iff the corresponding path ends at a final state.

In this context it is easy to see that the requirement that $\delta$ be total does not effect the class of languages accepted by DFAs; we can always extend a partial transition function to a total one by adding a *sink state* (such as state 3

[1] Indeed, it is more common to present DFAs in this way. We have chosen to present them in terms of IDs and computations in order to emphasize a consistent pattern from DFAs through PDAs and beyond.

in the example) which is non-final and from which no path ever leaves. Any edges with no place else to go can simply fall into the sink.

It should be reasonably clear that computations of a DFA correspond, in a very close way, to paths through its transition graph. Thus, if we are to prove lemmas about the set of strings accepted by a DFA, and hence about the set of paths through its transition graph, we will need a precise definition of the *path function*—the function that, given a starting node and a string, returns the ending node of the path from that starting node that is labeled with that string . This, as should come as no surprise, is defined inductively from $\delta$, the "edge" function.

**Definition 31 (Path Function of a DFA)** *The* path function $\hat{\delta} : Q \times \Sigma^* \to Q$ *is the extension of* $\delta$ *to strings:*

- *Basis:* $\hat{\delta}(q, \varepsilon) = q$, *for all* $q \in Q$.
- *Induction: If* $q \in Q$, $w \in \Sigma^*$ *and* $\sigma \in \Sigma$ *then* $\hat{\delta}(q, w\sigma) = \delta(\hat{\delta}(q, w), \sigma)$.

This just says that from any node the path of zero length (labeled $\varepsilon$) never leaves that node and that the ending node of the path from state $q$ labeled '$w\sigma$' can be found by first following the path from state $q$ labeled '$w$' and then following the edge labeled '$\sigma$'.

You should note that $\hat{\delta}(q, \sigma) = p$ (interpreting $\sigma$ as a unit string) iff $\delta(q, \sigma) = p$ (interpreting $\sigma$ as a symbol).

Just as the directly computes relation captures, in essence, the meaning of $\delta$, we can think of $\hat{\delta}$ as expressing the computes relation:

$$\hat{\delta}(q, w) = p \text{ iff } \langle q, w \rangle \vdash^* \langle p, \varepsilon \rangle .$$

We can use this to formalize what it means for an automaton to accept a string purely in terms of $\hat{\delta}$.

**Definition 32 (Language Accepted by a DFA (in terms of paths))** *The language* accepted *by a DFA* $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ *is*

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}.$$

This gives us a purely declarative definition of what it means for a given DFA to accept a given string. While we have motivated it in terms of the behavior of a particular sort of machine and in terms of certain graphs, it

does not in any way depend on those interpretations. $L(\mathcal{A})$ is defined purely in terms of $\hat{\delta}$, $q_0$ and $F$, and $\hat{\delta}$ is defined purely in terms of $\delta$. This is the definition we will use in proving claims about $L(\mathcal{A})$. While, again, we may motivate our proofs by appealing to machines or graphs, the actual proof itself will be in terms of the definitions of $L(\mathcal{A})$ and $\hat{\delta}$.

20. Sketch a proof that if $w \in L(\mathcal{A})$ according to Definition 29 then $w \in L(\mathcal{A})$ according to Definition 32. (Just give the base case(s), the IH, and an outline of the inductive step.)
[**Hint:** Start out by proving that $\langle q, w\rangle \vdash_{\mathcal{A}}^{*} \langle p, \varepsilon\rangle$ only if $\hat{\delta}(q, w) = p$.]

21. Sketch a proof of the converse: that if $w \in L(\mathcal{A})$ according to Definition 32 then $w \in L(\mathcal{A})$ according to Definition 29.
[**Hint:** Start with a lemma similar to that of the previous hint.]

22. Prove for all DFAs $\mathcal{A}$, that $\varepsilon \in L(\mathcal{A}) \Leftrightarrow q_0 \in F$. (Do not forget that you must prove both directions of the '$\Leftrightarrow$'.)

23. Our interest, in defining DFAs, is in defining $L(\mathcal{A})$. But $L(\mathcal{A})$ is defined in terms of $\hat{\delta}$ rather than $\delta$. Why, then, don't we define DFAs in terms of $\hat{\delta}$ instead of $\delta$?

24. Suppose $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ is a DFA and that $\hat{\delta}$ is defined accordingly. Prove that, for any strings $x$ and $y$ in $\Sigma^*$,

$$\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y).$$

[Hint: use induction on $|y|$.]

## 8.3 Defining Languages with DFAs

In this section we will develop a methodology for defining DFAs in a way that makes proving their correctness nearly automatic (although still somewhat tedious). There will usually be ways in which the proofs can be simplified, this methodology will always work. More importantly, in successfully defining a DFA you will inevitably have to carry out the first few steps of the method, even though you might do so implicitly. Thus the methodology is, in any case, a reasonable way to organize your attack on the problem.

**Example:** [Scheduling a machine tool] Consider the problem of specifying schedules for a machine tool which is used to manufacture a number of distinct types of parts (e.g., A, B, ...) and where each type of part may require a number of distinct operations (e.g., $A_1$, $A_2$, $A_3$, $B_1$, $B_2$, ...). Given a set of such operations, a *schedule* for the tool is a finite sequence of operations, i.e., a string in which the alphabet is just the set of operations. In general, there will be various constraints, such as restrictions on the order in which the operations are completed, etc. A *feasible schedule*, given some set of constraints, is a finite sequence of operations in which all the constraints are met *and* all parts are completed—the sequence has the same number of each of the operations required to complete a given type of part. We will return to variations of this problem later. For now, let us assume that there are two types of parts: A and B. Both require two operations: parts of type A require $A_1$ and $A_2$ and parts of type B require $B_1$ and $B_2$. These can be completed in any order, but, in a passing nod to realism, no more than two partially completed parts that can be stored. We will assume that whenever a part can be completed it will be, thus if the schedule calls for operation $A_1$, for instance, and there is any part which for which $A_2$ has been completed (but $A_1$ has not) then the operation will complete that part. A feasible schedule for this instance, then, is any string over $\{A_1, A_2, B_1, B_2\}$ in which

- the number of occurrences of $A_1$ is equal to the number of occurrences of $A_2$,
- the number of occurrences of $B_1$ is equal to the number of occurrences of $B_2$,
- in any initial segment of the sequence the difference between the number of '$A_1$'s and '$A_2$'s plus the difference between the number of '$B_1$'s and '$B_2$'s is never more than two.

Let $L_2$ be the language consisting of the set of such strings. Show that this is a regular set by providing an automaton and showing that it accepts all and only the strings in $L_2$.

A good way to attack a problem like this is to think of a DFA as a *classifier* of strings—all strings that label paths (from the start state) leading to the same state are lumped together. This is certainly true for the DFA, since all that it remembers about the portion of the input it has scanned is the state that it

has reached. If two strings lead to the same state then, from the DFA's point of view, they are indistinguishable.

The question for us is to figure out what information about the initial portion of a given string we need to keep track of in order to tell if the remainder of the string completes it in the sense of making it a feasible schedule. The key insight here is that all that we need to track is the type of any partially completed parts and whether at any point in the string there have ever been more than two parts pending. The remainder of the string will complete a feasible schedule iff it completes each of those outstanding parts without ever leaving more than two parts pending. Since the feasibility constraint is violated whenever more than two parts are pending, we never need to keep track of more than two partially completed parts. This is what bounds the amount of memory needed to recognize the language and is what makes it regular.

The state set, then, will include a state for each possible combination of two or fewer partially completed parts, plus a sink state for the case in which more than two are encountered at some point in the schedule. We will label these with the operations that have been completed on the pending parts:

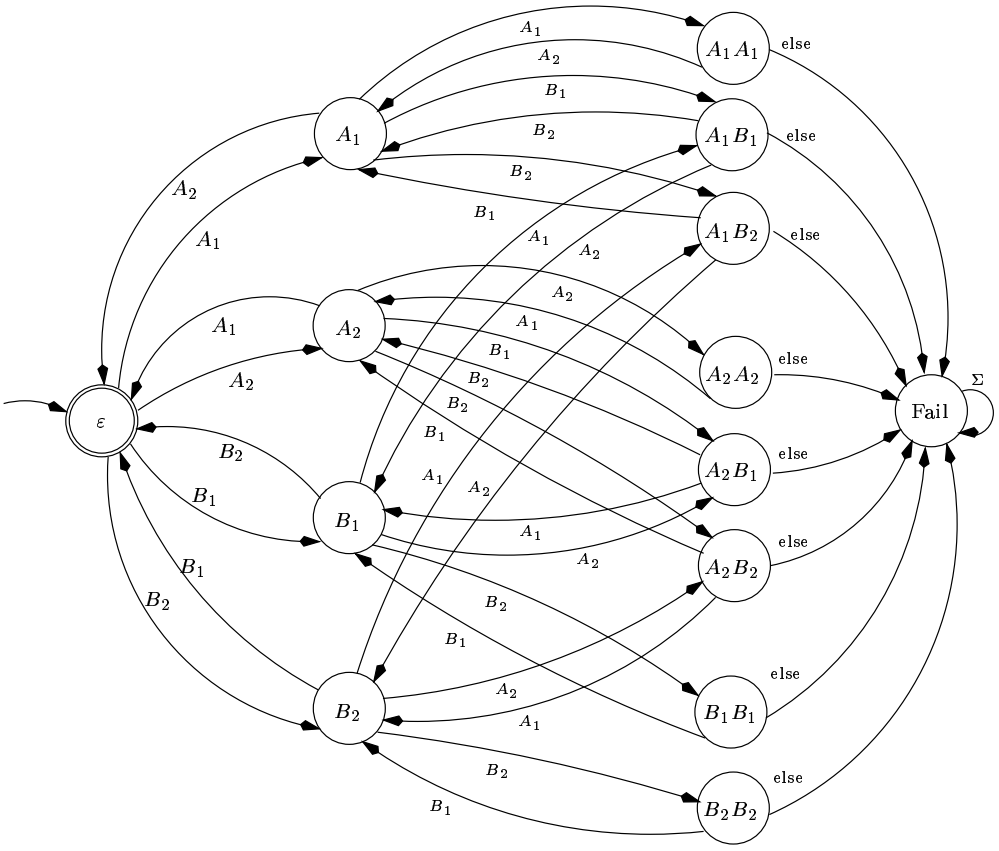
$$Q = \{\varepsilon, \mathrm{A}_1, \mathrm{A}_2, \mathrm{B}_1, \mathrm{B}_2, \mathrm{A}_1\mathrm{A}_1, \mathrm{A}_1\mathrm{B}_1, \mathrm{A}_1\mathrm{B}_2, \mathrm{A}_2\mathrm{A}_2, \mathrm{A}_2\mathrm{B}_1, \mathrm{A}_2\mathrm{B}_2, \mathrm{B}_1\mathrm{B}_1, \mathrm{B}_2\mathrm{B}_2, \mathrm{Fail}\}.$$

Note how we have obtained this state set: we start by identifying what characteristics of the strings we need to remember while scanning them and then define a state for each value those characteristics can take. The path from the start state labeled by any given string will lead to the state that encodes the distinguishing characteristics of that string. This is the only phase of this methodology that is not essentially automatic. It may well take a great deal of insight to be able to identify a set of characteristics that will work. Moreover, there will often be many ways one might do this for a given language and it will often be easier to prove that one state set properly characterizes the strings in a language than it will be to prove the same thing for another. Nonetheless, it is almost always harmless to distinguish more states than necessary—so long as you only distinguish finitely many of them—the only cost will be to make the rest of your proof longer.

In settling on a set of states and an interpretation of them in terms of the information they encode about the input strings we have fully determined the structure of the automaton. Transitions between states must preserve the interpretation of the states. If we are in some state '$q$' and see some input, '$\mathrm{A}_1$', for instance, the state we enter is determined by what happens if we

perform operation $A_1$ given the status of pending parts encoded by $q$. If $q$ is '$B_1$' we must enter state '$A_1B_1$'. If $q$ is '$A_2B_1$', on the other hand, we will complete the pending '$A$'—we go to state '$B_1$'. And if $q$ is '$A_1A_1$' we must fail. Furthermore, the start state must be that state encoding the status of the empty string—no partially completed parts in this case, state '$\varepsilon$'. Final states will be all states that encode strings that meet the specification of the language. Here this is all feasible schedules—those that complete every part without entering fail. These, then, will all end up in state '$\varepsilon$'; the set of final states is just $\{\varepsilon\}$.

So our choice of state sets yields the following automaton (call it $\mathcal{A}_2$):




We must now prove that $L(\mathcal{A}_2) = L_2$. To prove that every string accepted by the automaton is in $L_2$, i.e., that $L(\mathcal{A}_2) \subseteq L_2$, we must prove that every string labeling a path from '$\varepsilon$' that ends in a final state (i.e., in '$\varepsilon$') is in $L_2$. We actually do this by induction on the length of the path but, since there is a direct correspondence between paths in strings, this is the same as proving

it by induction on the length of the string. To prove that every string in $L_2$ is accepted by $\mathcal{A}_2$ (that $L_2 \subseteq L(\mathcal{A}_2)$) we must prove that every such string labels a path from '$\varepsilon$' back to '$\varepsilon$'. One way to do this would be by induction on the length of the string but this is very tedious. The approach we will take exploits the fact that our DFAs are total. Since every string labels a path from the start state to some state, if a string is not in the language accepted by the automaton it must label a path that ends at a non-final state. We can then argue that

$$w \in L_2 \Rightarrow w \in L(\mathcal{A}_2)$$

by the *contrapositive*

$$w \notin L(\mathcal{A}_2) \Rightarrow w \notin L_2,$$

which we can establish by showing that all strings that label paths from the start state that do not end in a final state are not in $L_2$.

What we need, then, is a characterization, for each state, of the strings that label paths to those states that is in terms that will allow us to show those strings are or are not in $L_2$. That is to say, we are looking to establish a system of *invariants*, one for each state, which suffice to prove the correctness of the automaton. These have the form:

$$\hat{\delta}(q_0, w) = q \Rightarrow \langle \text{Some characterization of } w \rangle,$$

for each $q \in Q$. (Note that, because we are arguing the backwards direction using the contrapositive, we need only prove the forward implication.)

We state the invariants in a compact form. Because of the way we have named the states, for all states other than the fail state the unmatched symbols in the string labeling the path to that state are exactly the symbols in the name of the state. Thus $|w|_{A_1} - |w|_{A_2} = |q|_{A_1} - |q|_{A_2}$,[2] i.e., the difference between the number of '$A_1$'s and '$A_2$'s is the same for both the string and the name of the state. The same is true for '$B$'s.

**Lemma 6 (Invariants)** *For all $w \in \Sigma^*$:*

(Fail:) $\hat{\delta}(\varepsilon, w) = \text{Fail} \Rightarrow w = uv$ *for some $u, v$ such that:*

$$\big||u|_{A_1} - |u|_{A_2}\big| + \big||u|_{B_1} - |u|_{B_2}\big| > 2.$$

[2] $|w|_\sigma$ is the number of occurrences of the symbol $\sigma$ in the string $w$.

**($q \neq$ Fail:)** $\hat{\delta}(\varepsilon, w) = q \neq \text{Fail} \Rightarrow$

$$|w|_{A_1} - |w|_{A_2} = |q|_{A_1} - |q|_{A_2}$$
$$|w|_{B_1} - |w|_{B_2} = |q|_{B_1} - |q|_{B_2}$$

*and for all prefixes $u$ of $w$*

$$\big||u|_{A_1} - |u|_{A_2}\big| + \big||u|_{B_1} - |u|_{B_2}\big| \leq 2.$$

**Proof:** [by induction on $|w|$]

(Basis:)

Suppose $|w| = 0$. Then $w = \varepsilon$, $\hat{\delta}(\varepsilon, \varepsilon) = \varepsilon$ and

$$|\varepsilon|_{A_1} = |\varepsilon|_{A_2} = |\varepsilon|_{B_1} = |\varepsilon|_{B_2} = 0.$$

One can verify that this satisfies the invariants, since $\hat{\delta}(\varepsilon, w) = \varepsilon \neq$ Fail and $w = \varepsilon$ (and, thus, have equal numbers of '$A_i$'s and '$B_i$'s).

(Inductive Step:)

Suppose that $|w| > 0$ and that the lemma is true for all strictly shorter strings. Then $w = w'\sigma$ for some $\sigma \in \Sigma$ and $w' \in \Sigma^*$ for which the lemma is true. To show that the lemma is true of $w$ as well:

To conserve space we will work with a table. (See Figure 1.)

The way the table should be read is:

> *If the entry in the first column is true then one of the associated rows must be the case, where the second column is the state after reading $w'$ and the third is the value of $\sigma$, the fourth and fifth record excess parts after $w'$ the sixth whether there are have been more than two parts pending after any (not necessarily proper) prefix of $w'$. and the seventh, eighth and ninth record the same information for $w$.*

The second and third column are taken from the definition of $\delta$, the fourth through sixth from the induction hypothesis, and the seventh through ninth from combining the previous entries in the rows.

Note that the inductive step for any one of the invariants depends on the induction hypothesis for one or more of the other invariants. Thus, they all must be proved together. Such a proof of a number of interdependent claims

| $\hat{\delta}(\varepsilon, w)$ | $\hat{\delta}(\varepsilon, w')$ | $\sigma$ | $\lvert w'\rvert_{A_1} - \lvert w'\rvert_{A_2}$ | $\lvert w'\rvert_{B_1} - \lvert w'\rvert_{B_2}$ | $w'$ Ovr | $\lvert w\rvert_{A_1} - \lvert w\rvert_{A_2}$ | $\lvert w\rvert_{B_1} - \lvert w\rvert_{B_2}$ | $w$ Ovr |
|---|---|---|---|---|---|---|---|---|
| $\varepsilon$ | $A_1$ | $A_2$ | 1 | 0 | × | 0 | 0 | × |
| | $A_2$ | $A_1$ | −1 | 0 | × | 0 | 0 | × |
| | $B_1$ | $B_2$ | 0 | 1 | × | 0 | 0 | × |
| | $B_2$ | $B_1$ | 0 | −1 | × | 0 | 0 | × |
| $A_1$ | $\varepsilon$ | $A_1$ | 0 | 0 | × | 1 | 0 | × |
| | $A_1A_1$ | $A_2$ | 2 | 0 | × | 1 | 0 | × |
| | $A_1B_1$ | $B_2$ | 1 | 1 | × | 1 | 0 | × |
| | $A_1B_2$ | $B_1$ | 1 | −1 | × | 1 | 0 | × |
| $A_2$ | $\varepsilon$ | $A_2$ | 0 | 0 | × | −1 | 0 | × |
| | $A_2A_2$ | $A_1$ | −2 | 0 | × | −1 | 0 | × |
| | $A_2B_1$ | $B_2$ | −1 | 1 | × | −1 | 0 | × |
| | $A_2B_2$ | $B_1$ | −1 | −1 | × | −1 | 0 | × |
| $B_1$ | $\varepsilon$ | $B_1$ | 0 | 0 | × | 0 | 1 | × |
| | $A_1B_1$ | $A_2$ | 1 | 1 | × | 0 | 1 | × |
| | $A_2B_1$ | $A_1$ | −1 | 1 | × | 0 | 1 | × |
| | $B_1B_1$ | $B_2$ | 0 | 2 | × | 0 | 1 | × |
| $B_2$ | $\varepsilon$ | $B_2$ | 0 | 0 | × | 0 | −1 | × |
| | $A_1B_2$ | $A_2$ | 1 | −1 | × | 0 | −1 | × |
| | $A_2B_2$ | $A_1$ | −1 | −1 | × | 0 | −1 | × |
| | $B_2B_2$ | $B_1$ | 0 | −2 | × | 0 | −1 | × |
| $A_1A_1$ | $A_1$ | $A_1$ | 1 | 0 | × | 2 | 0 | × |
| $A_1B_1$ | $A_1$ | $B_1$ | 1 | 0 | × | 1 | 1 | × |
| | $B_1$ | $A_1$ | 0 | 1 | × | 1 | 1 | × |
| $A_1B_2$ | $A_1$ | $B_2$ | 1 | 0 | × | 1 | −1 | × |
| | $B_2$ | $A_1$ | 0 | −1 | × | 1 | −1 | × |
| $A_2A_2$ | $A_2$ | $A_2$ | −1 | 0 | × | −2 | 0 | × |
| $A_2B_1$ | $A_2$ | $B_1$ | −1 | 0 | × | −1 | 1 | × |
| | $B_1$ | $A_2$ | 0 | 1 | × | −1 | 1 | × |
| $A_2B_2$ | $A_2$ | $B_2$ | −1 | 0 | × | −1 | −1 | × |
| | $B_2$ | $A_2$ | 0 | −1 | × | −1 | −1 | × |
| $B_1B_1$ | $B_1$ | $B_1$ | 0 | 1 | × | 0 | 2 | × |
| $B_2B_2$ | $B_2$ | $B_2$ | 0 | −1 | × | 0 | −2 | × |
| *Fail* | $A_1A_1$ | $A_1$ | 2 | 0 | × | 3 | 0 | ✓ |
| | | $B_1$ | 2 | 0 | × | 2 | 1 | ✓ |
| | | $B_2$ | 2 | 0 | × | 2 | −1 | ✓ |
| | $A_1B_1$ | $A_1$ | 1 | 1 | × | 2 | 1 | ✓ |
| | | $B_1$ | 1 | 1 | × | 1 | 2 | ✓ |
| | $A_1B_2$ | $A_1$ | 1 | −1 | × | 2 | −1 | ✓ |
| | | $B_2$ | 1 | −1 | × | 1 | −2 | ✓ |
| | $A_2A_2$ | $A_2$ | −2 | 0 | × | −3 | 0 | ✓ |
| | | $B_1$ | −2 | 0 | × | −2 | 1 | ✓ |
| | | $B_2$ | −2 | 0 | × | −2 | −1 | ✓ |
| | $A_2B_1$ | $A_2$ | −1 | 1 | × | −2 | 1 | ✓ |
| | | $B_1$ | −1 | 1 | × | −1 | 2 | ✓ |
| | $A_2B_2$ | $A_2$ | −1 | −1 | × | −2 | −1 | ✓ |
| | | $B_2$ | −1 | −1 | × | −1 | −2 | ✓ |
| | $B_1B_1$ | $A_1$ | 0 | 2 | × | 1 | 2 | ✓ |
| | | $A_2$ | 0 | 2 | × | −1 | 2 | ✓ |
| | | $B_1$ | 0 | 2 | × | 0 | 3 | ✓ |
| | $B_2B_2$ | $A_1$ | 0 | −2 | × | 1 | −2 | ✓ |
| | | $A_2$ | 0 | −2 | × | −1 | −2 | ✓ |
| | | $B_2$ | 0 | −2 | × | 0 | −3 | ✓ |
| | *Fail* | $\Sigma$ | — | — | ✓ | — | — | ✓ |

is called proof by *simultaneous induction*. Note also that even if all we wanted to establish was the invariant for '$\varepsilon$', we would need the invariants for '$A_1$', ..., '$B_2$' in order to carry out the inductive step of that proof and would, in turn, need the invariants for the other states (with the exception of 'Fail') in order to carry out those proofs. Thus, an invariant for each (non-sink) state is required in any event.

It is easy to verify from the table, by taking $q$ from the first column and the excess '$A_i$'s and '$B_i$'s in $w$ from the seventh and eighth, that the invariants hold in all cases. ⊣

There is nothing magical about the table, it simply is an essentially automatic way of organizing the tedious task of carrying out the proof exhaustively. Again, the argument can almost always be made in a more compact—read cleverer—fashion. But by working exhaustively you minimize the chance of overlooking cases.

It remains now to prove that the invariants imply that the automaton accepts a string iff it is a feasible schedule. This is nearly trivial. If $w \in L(\mathcal{A})$ then $\hat{\delta}(\varepsilon, w) = \varepsilon$ which implies, by the invariant, that the number of '$A_1$'s and '$A_2$'s and the number of '$B_1$'s and '$B_2$'s are equal and that there is no prefix of $w$ in which the sum of the absolute differences between these ever exceeded two. Conversely, if $w \notin L(\mathcal{A})$ then $\hat{\delta}(\varepsilon, w) \in Q \setminus \{\varepsilon\}$ (because $\delta$ and, consequently $\hat{\delta}$, is total). Again, from the invariants, it is easy to verify that this implies that there is either a partially completed part left at the end of the schedule or that the number of partially completed parts exceeded two at some point in the schedule.

25. Consider a system consisting of two processes (A and B) exchanging messages. Process A sends two types of messages to process B: $m_1$ and $m_2$. Process B sends three types of acknowledgment to process A: $a_1$, $a_2$, and $a_{12}$, which acknowledge $m_1$, $m_2$, and $m_1$ and $m_2$ simultaneously, respectively. The processes are required to follow the following protocol: Process A can send either message type at any time, but only if every prior message of that type has been acknowledged. Process B may send any acknowledge at any time, but only if it has received a message(s) of the appropriate type(s) which it has not yet acknowledged. Process B is required to eventually acknowledge every message received from Process A.

    Finite sequences of messages exchanged within this system are just strings

over the alphabet $\{m_1, m_2, a_1, a_2, a_{12}\}$. Let $L_3$ be the language consisting of the set of such strings in which the protocol sketched above is followed. Show that this is a regular set by providing an automaton and showing that it accepts all and only the strings in $L_3$.

e.g.,

$$\begin{aligned} m_2m_1a_2m_2a_2a_1m_1m_2a_{12} &\in L_3, \\ m_2m_1a_2\underline{m_1}a_2a_1m_1m_2a_{12} &\notin L_3, \\ m_2m_1a_2m_2a_2a_1m_1\underline{a_2}a_1 &\notin L_3, \\ m_2m_1a_2m_2a_2a_1\underline{m_1m_2} &\notin L_3. \end{aligned}$$

# 9 Non-Deterministic Finite-State Automata (NFAs)

## 9.1 Basic NFAs

Our DFAs are required to have transition functions that are total (so there is a next state for every current state and input symbol) and to return a single state (so there is a unique state for every current state and input symbol). Thus, the next state is fully determined by the current state and input symbol. As we saw in the previous section, this simplifies the proof that the DFA accepts a specific language. There are many circumstances, though, in which it will be simpler to define the automaton in the first place if we allow a choice of next state or even no next state. We will still require transitions to be defined by a function, but it will now return a *set of states* rather than a single state.

**Definition 33 (NFA without $\varepsilon$-Transitions)** *A* Non-deterministic Finite-state Automaton (NFA) (without $\varepsilon$-transitions) *is a 5-tuple:* $\langle Q, \Sigma, \delta, q_0, F\rangle$ *where:*

$Q$, $\Sigma$, $q_0$ *and* $F$ *are as in a DFA,*

$\delta : Q \times \Sigma \to \boldsymbol{\mathcal{P}}(Q)$ *is the* transition function

*(mapping a state and an input symbol to the set of choices of next state)*

Here $\boldsymbol{\mathcal{P}}(Q)$ denotes the powerset of $Q$—the set of all of its subsets. It should be emphasized that the transition function *is* still a function and is still total. We have accommodated the possibilities of there being either more than one or no potential next state by returning the set of next states: if $\mathbf{card}(\delta(q, \sigma)) = 1$, then the transition on $\langle q, \sigma\rangle$ is deterministic; if it is zero then there is no transition licensed for $\langle q, \sigma\rangle$ and the NFA will "crash"—halt prior to scanning the full input.

Instantaneous descriptions of NFAs are identical to those of DFAs.:

**Definition 34 (Instantaneous Description (for both DFAs and NFAs))** *An* instantaneous description *of* $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$*, either a DFA or an NFA, is a pair* $\langle q, w\rangle \in Q \times \Sigma^*$*, where* $q$ *the current state and* $w$ *is the portion of the input under and to the right of the read head.*

The directly computes relation is nearly also the same—we need only to account for the fact the transition function returns a set of states rather than a unique next state.

**Definition 35 (Directly Computes Relation (for NFAs without $\varepsilon$-transitions))**

$$\langle q, w\rangle \vdash_{\mathcal{A}} \langle p, v\rangle \Leftrightarrow w = \sigma v \text{ and } p \in \delta(q, \sigma).$$

Note that while this is defined essentially identically for both DFAs and NFAs, in the case of NFAs it is no longer even partial functional; an ID may well have many successors. Moreover, it is no longer true that then only IDs without successors are those in which $w = \varepsilon$.

The definition of computation is, again, identical for both NFAs and DFAs. We do, however, need to amend our notion of *closure under* $\vdash_{\mathcal{A}}$ to account for the fact that, while there may now be more than one successor of an ID, only one of those can actually follow it in a single computation.

**Definition 36 (Computation (for both DFA and NFAs))** *A computation of a DFA* $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ *from state* $q_1$ *on input* $w_1$ *is a sequence of IDs* $\langle \langle q_1, w_1\rangle, \ldots\rangle$ *in which, for all* $i > 0$, $\langle q_{i-1}, w_{i-1}\rangle \vdash_{\mathcal{A}} \langle q_i, w_i\rangle$ *and which is closed under* $\vdash_{\mathcal{A}}$ .

Here *closed under* $\vdash_{\mathcal{A}}$ means: for all $i$, if $\langle q_i, w_i\rangle$ has *any* successor, than *one* of those successors will be included in the sequence as $\langle q_{i+1}, w_{i+1}\rangle$. The fact that $\vdash_{\mathcal{A}}$ is no longer partial functional implies that we can no longer speak of *the* computation of $\mathcal{A}$ on $\langle q_1, w_1\rangle$. What's more, while every computation is finite, it is no longer true that they all take exactly $|w_1|$ steps. Computations end when they reach an ID with no successor; this can now be either because the entire input was scanned or because an ID $\langle q, \sigma \cdot w\rangle$ was reached for which $\delta(q, \sigma) = \emptyset$. Only the first case represents successful processing of $w_1$ by $\mathcal{A}$; we need to be careful to distinguish "halting" computations from those that "crash".

Nevertheless, neither the computes relation nor the definition of the language accepted by the automaton need be modified:

**Definition 37 (Computes Relation (for both DFAs and NFAs))** $\vdash^*_{\mathcal{A}}$ *is the reflexive, transitive closure of* $\vdash_{\mathcal{A}}$ .

**Definition 38 (Language Accepted by a DFA or NFA)** *The* language accepted by a DFA $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ *is*

$$L(\mathcal{A}) = \{w \mid \langle q_0, w\rangle \vdash^*_{\mathcal{A}} \langle q, \varepsilon\rangle, q \in F\}$$

The fact that we require the final ID of the computation accepting $w \in L(\mathcal{A})$ to be of the form $\langle q, \varepsilon \rangle$ means that computations that crash before scanning all of the input are ruled out.

**Example:** For example:

| | | $Q$ | | | |
|---|---|---|---|---|---|
| | $\delta$ | 0 | 1 | 2 | 3 |
| $\Sigma$ | $a$ | $\{1,3\}$ | $\{2,3\}$ | $\{3\}$ | $\{3\}$ |
| | $b$ | $\emptyset$ | $\{0,1,2,3\}$ | $\emptyset$ | $\{0,2\}$ |

$F = \{0,2\}$

The language accepted includes the string '$abb$' as witnessed by the computation

$$\langle \langle 0, abb \rangle, \langle 1, bb \rangle, \langle 3, b \rangle, \langle 0, \varepsilon \rangle \rangle.$$

There are also other accepting computations on '$abb$':

$$\langle \langle 0, abb \rangle, \langle 1, bb \rangle, \langle 1, b \rangle, \langle 0, \varepsilon \rangle \rangle$$
$$\langle \langle 0, abb \rangle, \langle 1, bb \rangle, \langle 1, b \rangle, \langle 2, \varepsilon \rangle \rangle$$

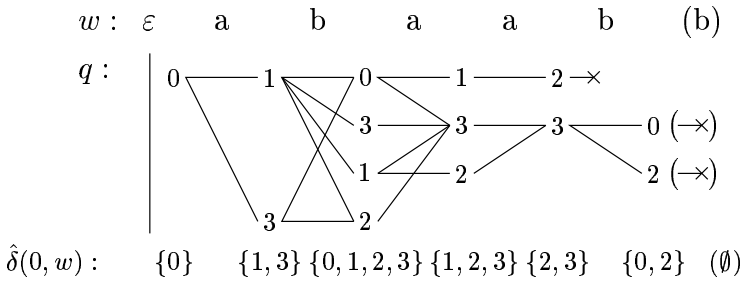
as well as computations that fail to accept '$abb$':

$$\langle \langle 0, abb \rangle, \langle 1, bb \rangle, \langle 1, b \rangle, \langle 3, \varepsilon \rangle \rangle$$
$$\langle \langle 0, abb \rangle, \langle 1, bb \rangle, \langle 2, b \rangle \rangle$$

26. Give two distinct accepting computations of the NFA of the previous example on the string '$abab$'.

27. Give two distinct non-accepting computations of the NFA of the previous example on the string '$abab$'.

NFAs correspond to a kind of parallelism in the automata. We can think of the same basic model of automaton: an input tape, a single read head and an internal state, but when the transition function allows more than one next state for a given state and input we keep an independent internal state for each of the alternatives. In a sense we have a constantly growing and shrinking set of automata all processing the same input synchronously. For example, a computation of the NFA given above on '$abaab$' could be interpreted as:

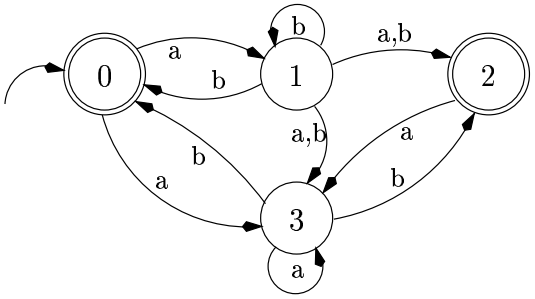



This string is accepted, since there is at least one computation from 0 to 0 or 2 on '*abaab*'. Similarly, each of '$\varepsilon$', '*ab*', '*aba*' and '*abaa*' are accepted, but '*a*' alone is not. Note that if the input continues with '*b*' as shown there will be no states left; the automaton will crash. Clearly, it can accept no string starting with '*abaabb*' since the computations from 0 or '*abaabb*' end either in $\langle 0, b\rangle$ or in $\langle 2, b\rangle$ and, consequentially, so will all computations from 0 on any string extending it. The fact that in this model there is not necessarily a (non-crashing) computation from $q_0$ for each string complicates the proof of the language accepted by the automaton—we can no longer assume that if there is no (non-crashing) computation from $q_0$ to a final state on $w$ then there must be a (non-crashing) computation from $q_0$ to a non-final state on $w$. As we shall see, however, we will never need to do such proofs for NFAs directly.

## 9.2 Transition Graphs of NFAs (without $\varepsilon$-transitions)

In terms of the transition graph of the automaton, if $\delta(q, \sigma) = \{q_1, q_2, \ldots, q_k\}$ then there will be an edge labeled $\sigma$ from state $q$ to each of the $q_1, q_2, \ldots, q_k$. This means that if there is a path labeled $w$ leading from some $q_0$ to $q$, then there are paths labeled $w\sigma$ from $q_0$ to each of the $q_1, q_2, \ldots, q_k$. In the case that $\delta(q, \sigma) = \emptyset$ there is no edge labeled $\sigma$ from state $q$ in the transition graph and no path labeled $w\sigma$ that visits $q$ in its next-to-last state.

**Example:** The transition graph of the NFA of the example of the last section is given in Figure 2.




| | | $Q$ | | | |
|---|---|---|---|---|---|
| | $\delta$ | 0 | 1 | 2 | 3 |
| $\Sigma$ | $a$ | $\{1,3\}$ | $\{2,3\}$ | $\{3\}$ | $\{3\}$ |
| | $b$ | $\emptyset$ | $\{0,1,2,3\}$ | $\emptyset$ | $\{0,2\}$ |

$F = \{0,2\}$

Figure 2: An NFA and its transition graph.

Clearly, the path function also needs to return sets of states rather than states:

**Definition 39 (Path Function of a NFA (preliminary version))** *The* path function $\hat{\delta} : Q \times \Sigma^* \to \boldsymbol{\mathcal{P}}(Q)$ *is the extension of* $\delta$ *to strings:*

- *Basis:* $\hat{\delta}(q, \varepsilon) = \{q\}$*, for all* $q \in Q$*.*
- *Induction: If* $q \in Q$*,* $w \in \Sigma^*$ *and* $\sigma \in \Sigma$ *then* $\hat{\delta}(q, w\sigma) = \bigcup_{q' \in \hat{\delta}(q,w)} [\delta(q', \sigma)]$*.*
- *Nothing else.*

This just says that the path labeled $\varepsilon$ from any given state $q$ goes only to $q$ itself (or rather never leaves $q$) and that to find the set of states reached by paths labeled $w\sigma$ from $q$ one first finds all the states $q'$ reached by paths labeled $w$ from $q$ and then takes the union of all the states reached by an edge labeled $\sigma$ from any of those $q'$. Another way of expressing it is

$$\hat{\delta}(q, w\sigma) = \{q'' \mid (\exists q' \in \hat{\delta}(q, w))[q'' \in \delta(q', \sigma)]\}.$$

We will still accept a string $w$ iff there is a path labeled $w$ leading from the initial state to a final state, but now there may be many paths labeled $w$

from the initial state, some of which reach final states and some of which do not. In the context of transition graphs, we need to modify the definition of the language accepted by $\mathcal{A}$ so it includes every string for which at least one path ends at a final state.

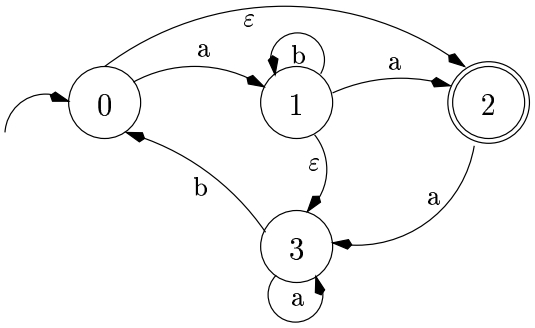
**Definition 40 (Language Accepted by a NFA)** *The language* accepted *by a NFA* $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ *is*

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}.$$

## 9.3 NFAs with $\varepsilon$-Transitions

We now add an additional degree of non-determinism and allow transitions that can be taken independent of the input—*$\varepsilon$-transitions.*

**Example:**




Here whenever the automaton is in state 1 it may make a transition to state 3 *without consuming any input.* Similarly, if it is in state 0 it may make such a transition to state 2. The advantage of such transitions is that they allow one to build NFAs in pieces, with each piece handling some portion of the language, and then splice the pieces together to form an automaton handling the entire language. To accommodate these transitions we need to modify the type of the transition function to map pairs drawn from $Q$ and $\Sigma \cup \{\varepsilon\}$ to subsets of $Q$.

**Definition 41 (NFA with $\varepsilon$-Transitions)** *A* Non-deterministic Finite-state Automaton (NFA) (with $\varepsilon$-transitions) *is a 5-tuple:* $\langle Q, \Sigma, \delta, q_0, F \rangle$ *where* $Q$, $\Sigma$, $q_0$ *and* $F$ *are as in a DFA and* $\delta$ *is of the type* $Q \times (\Sigma \cup \{\varepsilon\}) \to \boldsymbol{\mathcal{P}}(Q)$, *where* $\boldsymbol{\mathcal{P}}(Q)$ *denotes the* power set *of* $Q$, *i.e., the set of all subsets of* $Q$.

We must also modify the definitions of the directly computes relation and the path function to allow for the possibility that $\varepsilon$-transitions may occur anywhere in a computation or path. The $\varepsilon$-transition from state 1 to state 3 in the example, for instance, allows the automaton on input '$a$' to go from state 0 not only to state 1 but also to immediately go to state 3. Similarly, it allows the automaton, when in state 1 with input '$b$', to move first to state 3 and then take the '$b$' edge to state 0 or, when in state 0 with input '$a$', to move first to state 2 and then take the '$a$' edge to state 3. Thus, on a given input '$\sigma$', the automaton can take any sequence of $\varepsilon$-transitions followed by exactly one $\sigma$-transition and then any sequence of $\varepsilon$-transitions. To capture this in the definition of $\hat{\delta}$ we start by defining the function $\varepsilon$-Closure which, given a state, returns the set of all states reachable from it by any sequence of $\varepsilon$-transitions.

**Definition 42 ($\varepsilon$-Closure of a State)** *The* $\varepsilon$-Closure *of a state* $q$ *of an automaton* $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ *is defined inductively as follows:*

- *Basis:* $q \in \varepsilon\text{-Closure}(q)$.
- *Ind: If* $q' \in \varepsilon\text{-Closure}(q)$ *then* $\delta(q', \varepsilon) \subseteq \varepsilon\text{-Closure}(q)$
- *Nothing else.*

*The* $\varepsilon$-Closure *of a set of states* $S \subseteq Q$ *is*

$$\varepsilon\text{-Closure}(S) \stackrel{\text{def}}{=} \bigcup_{q \in S} [\varepsilon\text{-Closure}(q)].$$

With this we can modify the definitions of directly computes and the path function.

**Definition 43 (Directly Computes Relation (for NFAs in general))**

$$\langle q, w\rangle \underset{\mathcal{A}}{\vdash} \langle p, v\rangle \Leftrightarrow w = \sigma v \text{ and } p \in \varepsilon\text{-Closure}(\delta(\varepsilon\text{-Closure}(q), \sigma)).$$

**Definition 44 (Path Function of a NFA (final version))** *The* path function $\hat{\delta} : Q \times \Sigma^* \to \boldsymbol{\mathcal{P}}(Q)$ *is the extension of* $\delta$ *to strings:*

- *Basis:* $\hat{\delta}(q, \varepsilon) = \varepsilon\text{-Closure}(q)$, *for all* $q \in Q$.
- *Ind: If* $q \in Q$, $w \in \Sigma^*$ *and* $\sigma \in \Sigma$
  *then* $\hat{\delta}(q, w\sigma) = \bigcup_{q' \in \hat{\delta}(q,w)} [\varepsilon\text{-Closure}(\delta(q', \sigma))]$.

- *Nothing else.*

In essence, these say that the set of states that may reach from ID $\langle q, \sigma \cdot w \rangle$ are those that can be reached by making any number of $\varepsilon$-transitions and exactly one $\sigma$ transition, and that to find the set of states reachable by a path labeled $w$ from a state $q$ in an NFA with $\varepsilon$-transitions start by finding the set of states reachable from $q$ using only $\varepsilon$-transitions and then, for each symbol $\sigma$ of $w$ (in order) find the set of states reachable from those by an edge labeled $\sigma$ and then the set of states reachable from *those* by any sequence of $\varepsilon$-transitions, etc.

Nothing else in the definitions need change. The automaton still accepts $w$ if there is any computation on $\langle q_0, w \rangle$ that terminates in a final state after scanning the entire input. Equivalently, it accepts $w$ if there is a path labeled $w$ from the initial state to a final state, which is to say, if $\hat{\delta}(q_0, w)$ includes any member of $F$. Note that the automaton of Example 9.3 will accept '$\varepsilon$' since state 2 is in $\varepsilon$-Closure(0) and, therefore in $\hat{\delta}(0, \varepsilon)$.

## 9.4 Equivalence of NFAs with and without $\varepsilon$-transitions

It is not hard to see that $\varepsilon$-transitions do not add to the accepting power of the model. The effect of the $\varepsilon$-transition from state 1 to state 3 in the example, for instance, can be obtained by adding '$a$' edges from 0 and 1 to 3 and a '$b$' edge from 1 to 3. Similarly, most of the effect of the $\varepsilon$-transition from 0 to 2 can be obtained by adding an '$a$' transition from 0 to 3 and '$b$' transitions from 1 and 3 to 2. Note that in both these cases this corresponds to extending $\delta(q, \sigma)$ to include all states in $\hat{\delta}(q, \sigma)$. The remaining effect of the $\varepsilon$-transition from 0 to 2 is the fact that the automaton accepts '$\varepsilon$'. This can be obtained, of course, by simply adding 0 to $F$. Formalizing this we get a lemma.

**Lemma 7 (Equivalence of NFAs with and without $\varepsilon$-transitions)** *A language $L \subseteq \Sigma^*$ is $L(\mathcal{A})$ for an NFA with $\varepsilon$-transitions $\mathcal{A}$ iff it is $L(\mathcal{A}')$ for an NFA without $\varepsilon$-transitions $\mathcal{A}'$.*

**Proof:** Note, first, that every NFA without $\varepsilon$-transitions is trivially an NFA with (no) $\varepsilon$-transitions as well. For the other direction we will show how, given an NFA with $\varepsilon$-transitions, to construct an NFA without $\varepsilon$-transitions

that accepts the same language.
For $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ let $\mathcal{A}' = \langle Q, \Sigma, \delta', q_0, F'\rangle$ where

$$\begin{aligned} \delta'(q,\sigma) &= \hat{\delta}(q,\sigma), \quad \text{for all } q \in Q \text{ and } \sigma \in \Sigma \\ \text{and} \quad F' &= \begin{cases} F \cup \{q_0\} & \text{if } \hat{\delta}(q_0,\varepsilon) \cap F \neq \emptyset, \\ F & \text{otherwise .} \end{cases} \end{aligned}$$

Note that $\delta'$ includes no $\varepsilon$-transitions and, therefore, $\mathcal{A}'$ is the appropriate type of NFA. To show that $L(\mathcal{A}') = L(\mathcal{A})$ we need to establish that

$$\hat{\delta}'(q_0, w) \cap F' \neq \emptyset \Leftrightarrow \hat{\delta}(q_0, w) \cap F \neq \emptyset.$$

This would, of course, follow if we could show that $\hat{\delta}'(q, w) = \hat{\delta}(q, w)$ for all $q$ and $w$, and it is not hard to see that for non-empty $w$ this will be the case. The problem, of course, is that it will not be the case for $w = \varepsilon$ since, by definition,

$$\hat{\delta}'(q, \varepsilon) = \{q\}$$

while

$$\hat{\delta}(q, \varepsilon) = \varepsilon\text{-Closure}(q).$$

But the '$\varepsilon$' case is handled by the fact that $q_0$ has been added to $F'$ if $\hat{\delta}(q_0, \varepsilon) \cap F \neq \emptyset$. Thus,

$$\begin{aligned} \varepsilon \in L(\mathcal{A}') &\Leftrightarrow \hat{\delta}'(q_0,\varepsilon) \cap F' \neq \emptyset \\ &\Leftrightarrow q_0 \in F' \\ &\Leftrightarrow q_0 \in F \text{ or } \hat{\delta}(q_0,\varepsilon) \cap F \neq \emptyset \\ &\Leftrightarrow \hat{\delta}(q_0,\varepsilon) \cap F \neq \emptyset \qquad \text{since } q_0 \in \hat{\delta}(q_0,\varepsilon) \\ &\Leftrightarrow \varepsilon \in L(\mathcal{A}). \end{aligned}$$

What remains, then, is to establish the $\hat{\delta}$ and $\hat{\delta}'$ coincide on all strings of length at least one. This is left to you as an exercise.

28. Prove the claim: for all $w \in \Sigma^+$ and $q \in Q$, $\hat{\delta}'(q, w) = \hat{\delta}(q, w)$.

⊣

## 9.5 Equivalence of NFAs and DFAs

In general non-determinism, by introducing a degree of parallelism, may increase the accepting power of a model of computation. But if we subject NFAs to the same sort of analysis as we have used in defining DFAs we shall see that to simulate an NFA one needs only track finitely much information about each string. Consider, again, the example in which we modeled the computation of the NFA as a set of automata processing the input synchronously. In order to determine if a string $w$ is accepted by the NFA all we need to do is to track, at each stage of the computation (i.e., at each prefix of the input), the states of those automata. Since there is never any reason to include more than one automaton for each state, this will just be some subset of $Q$—in fact, it is easy to see that the set of states after processing $w$ will be just $\hat{\delta}(q_0, w)$. Since $Q$ is finite, it has finitely many subsets. Thus we can simulate an NFA with state set $Q$ with a DFA that has a state for each subset of $Q$. The process of constructing a deterministic analog of a non-deterministic machine is known as *determinization.*

**Lemma 8 (Equivalence of NFAs and DFAs)** *A language $L \subseteq \Sigma^*$ is $L(\mathcal{A})$ for an NFA (with or without $\varepsilon$-transitions) $\mathcal{A}$ iff it is $L(\mathcal{A}')$ for some DFA $\mathcal{A}'$.*

**Proof** (Subset Construction)**:** Again, if $L = L(\mathcal{A}')$ for a DFA it is easy to see that it is also $L(\mathcal{A})$ for an NFA—one in which the transition function always returns a singleton set of states. For the other direction we, again, will give a construction, this time one that, given a NFA, builds a DFA accepting the same language. Because the state set of the DFA will be the set of all subsets of the state set of the NFA this construction is known as the *subset construction.* For $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ an NFA (for ease of proof we will assume that it does not have $\varepsilon$-transitions) let $\mathcal{A}' = \langle Q', \Sigma, Q'_0, \delta', F'\rangle$, where:

$$\begin{aligned} Q' &= \boldsymbol{\mathcal{P}}(Q) \\ Q'_0 &= \{q_0\} \\ \delta(S,\sigma) &= \bigcup_{q\in S}[\delta(q,\sigma)] \qquad \text{for each } \sigma \in \Sigma \text{ and } S \in Q' (\text{i.e., } S \subseteq Q) \\ F' &= \{S \subseteq Q \mid S \cap F \neq \emptyset\}. \end{aligned}$$

We must prove for all $w \in \Sigma^*$ that

$$\hat{\delta}'(Q'_0, w) \cap F' \neq \emptyset \Leftrightarrow \hat{\delta}(q_0, w) \cap F \neq \emptyset.$$

We will do this by proving the slightly strengthened claim:

$$\text{for all } w \in \Sigma^*, \qquad \hat{\delta}'(Q_0', w) = \hat{\delta}(q_0, w),$$

by induction on $|w|$.

(Basis)

Suppose $|w| = 0$. Then $w = \varepsilon$ and

$$\hat{\delta}'(Q_0', \varepsilon) = \{q_0\} = \delta(q_0, \varepsilon).$$

(Induction)

Suppose $|w| = n + 1$ and that the claim is true for all strings of length $n$. Then $w = v\sigma$ for some $\sigma \in \Sigma$ and $v \in \Sigma^*$ of length $n$. To show that the claim is true for $w$ as well:

$$\begin{aligned}
\hat{\delta}'(Q_0', v\sigma) &= \delta'(\hat{\delta}'(Q_0', v), \sigma) && \text{by definition of } \hat{\delta}' \\
&= \delta'(\hat{\delta}(q_0, v), \sigma) && \text{by IH} \\
&= \bigcup_{q' \in \hat{\delta}(q_0, v)} [\delta(q', \sigma)] && \text{by definition of } \delta' \\
&= \hat{\delta}(q_0, v\sigma) && \text{by definition of } \hat{\delta}.
\end{aligned}$$

Then

$$\begin{aligned}
w \in L(\mathcal{A}') &\Leftrightarrow \hat{\delta}'(Q_0', w) \in F' \\
&\Leftrightarrow \hat{\delta}'(Q_0', w) \cap F \neq \emptyset \\
&\Leftrightarrow \hat{\delta}(q_0, w) \cap F \neq \emptyset \\
&\Leftrightarrow w \in L(\mathcal{A}).
\end{aligned}$$

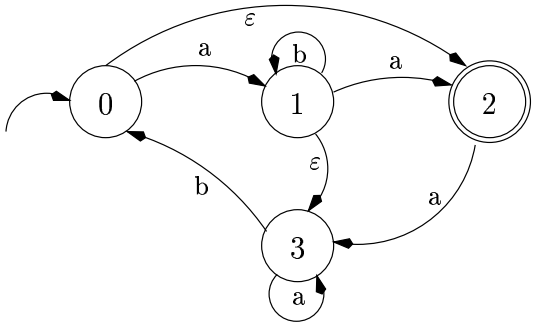
⊣

### 9.5.1 Applying Subset Construction

As defined the subset construction builds a DFA with many states that can never be reached from $Q_0'$. Since they cannot be reached from $Q_0'$ there is no path from $Q_0'$ to a state in $F'$ which passes through them and they can be deleted from the automaton without changing the language it accepts. In

practice it is much easier to build $Q'$ as needed, only including those state sets that actually are needed.

To see how this works, lets carry out an example. For maximum generality, let's start with the NFA *with $\varepsilon$-transitions* given above, repeated here:




When given a transition graph of an NFA with $\varepsilon$-transitions like this there are 6 steps required to reduce it to a DFA:

1. Write out the transition function and set of final states of the NFA.
2. Convert it to an NFA without $\varepsilon$-transitions.
   (a) Compute the $\varepsilon$-Closure of each state in the NFA.
   (b) Compute the transition function of the equivalent NFA without $\varepsilon$-transitions.
   (c) Compute the set of final states of the equivalent NFA without $\varepsilon$-transitions.
3. Convert the equivalent NFA to a DFA.
   (a) Starting with $\{q_0\}$ and repeating for each state set encountered in the construction compute the transition function for each input symbol.
   (b) Compute the set of final states of the equivalent DFA.

While it is tempting to work directly from the transition graph and to combine steps there are two reasons to not do so. First, both shortcuts are prone to error. It is easy to miss edges when referring repeatedly to the graph and it is, in particular, easy to miss relevant $\varepsilon$-transitions when trying to simultaneously remove them and determinize the result. The second reason

is that we have only proved the correctness of the constructions eliminating $\varepsilon$-transitions and determinizing the result separately. While it is not difficult to define a construction combining them, if you use such a construction you must prove its correctness.

The transition function:

| $q$ | $\delta(q,a)$ | $\delta(q,b)$ | $\delta(q,\varepsilon)$ |
|---|---|---|---|
| 0 | $\{1\}$ | $\emptyset$ | $\{2\}$ |
| 1 | $\{2\}$ | $\{1\}$ | $\{3\}$ |
| 2 | $\{3\}$ | $\emptyset$ | $\emptyset$ |
| 3 | $\{3\}$ | $\{0\}$ | $\emptyset$ |

$F = \{2\}$.

$\varepsilon$-Closure:

| $q$ | $\varepsilon$-Closure(q) |
|---|---|
| 0 | $\{0,2\}$ |
| 1 | $\{1,3\}$ |
| 2 | $\{2\}$ |
| 3 | $\{3\}$ |

The transition function of the equivalent NFA without $\varepsilon$-transitions:

$$\delta'(q,\sigma) \stackrel{\text{def}}{=} \hat{\delta}(q,\sigma) \stackrel{\text{def}}{=} \bigcup_{q' \in \varepsilon\text{-Closure}(q)} [\varepsilon\text{-Closure}(\delta(q'\sigma))].$$

| $q$ | $\varepsilon$-Closure$(q)$ | $\delta'(q,a)$ | $\delta'(q,b)$ |
|---|---|---|---|
| 0 | $\{0,2\}$ | $\{1,3\}$ | $\emptyset$ |
| 1 | $\{1,3\}$ | $\{2,3\}$ | $\{0,1,2,3\}$ |
| 2 | $\{2\}$ | $\{3\}$ | $\emptyset$ |
| 3 | $\{3\}$ | $\{3\}$ | $\{0,2\}$ |

$F' = \{0,2\}$ since $\varepsilon$-Closure$(0) = \{0,2\} \cap F \neq \emptyset$.

To construct the transition function of the equivalent DFA start with $S = Q'_0 = \{q_0\} = \{0\}$ and compute

$$\delta''(S,\sigma) = \bigcup_{q \in S} [\delta'(q,\sigma)].$$

for each $\sigma \in \Sigma$.

| $S$ | $\delta''(S,a)$ | $\delta''(S,b)$ |
|---|---|---|
| $\{0\}$ | $\{1,3\}$ | $\emptyset$ |

Next introduce lines for each of these state sets

| $S$ | $\delta''(S,a)$ | $\delta''(S,b)$ |
|---|---|---|
| $\{0\}$ | $\{1,3\}$ | $\emptyset$ |
| $\{1,3\}$ | $\{2,3\}$ | $\{0,1,2,3\}$ |
| $\emptyset$ | $\emptyset$ | $\emptyset$ |

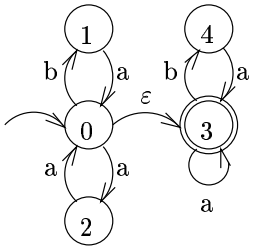
and repeat until no new lines are introduced

| $S$ | $\delta''(S,a)$ | $\delta''(S,b)$ |
|---|---|---|
| $\{0\}$ | $\{1,3\}$ | $\emptyset$ |
| $\{1,3\}$ | $\{2,3\}$ | $\{0,1,2,3\}$ |
| $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $\{2,3\}$ | $\{3\}$ | $\{0,2\}$ |
| $\{0,1,2,3\}$ | $\{1,2,3\}$ | $\{0,1,2,3\}$ |
| $\{3\}$ | $\{3\}$ | $\{0,2\}$ |
| $\{0,2\}$ | $\{1,3\}$ | $\emptyset$ |
| $\{1,2,3\}$ | $\{2,3\}$ | $\{0,1,2,3\}$ |

The set of final state $F''$ is then the set of state sets that include states of $F'$:

$$F'' = \{\{0\},\{2,3\},\{0,1,2,3\},\{0,2\},\{1,2,3\}\}.$$

29. Convert the following NFA to a DFA.



# 10 The Equivalence of DFAs and Regular Expressions

We have now looked at two ways of defining languages—as the denotation of a regular expression and as the set accepted by a DFA or, equivalently, an NFA. Surprisingly, the class of languages that can be defined in the one way turns out to be exactly the class of languages definable in the other. This theorem, originally due to Kleene, was one of the first dramatic theorems of Formal Language Theory. We will establish it in two parts: first we will show how to convert any regular expression to an equivalent NFA and then we will show how to convert any DFA into an equivalent regular expression. Since we have already shown the equivalence of NFAs and DFAs, this will suffice.

## 10.1 Constructing NFAs from Regular Expressions

**Lemma 9** *If $L \subseteq \Sigma^*$ is $L(R)$ for a regular expression $R$ then it is also $L(\mathcal{A})$ for an NFA.*

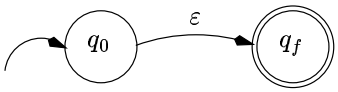
**Proof:** Not surprisingly, we will prove this by induction on the structure of the regular expression. To simplify the proof, we will strengthen the hypothesis slightly: IF $L = L(R)$ for a regular expression $R$ then $L = L(\mathcal{A})$ for an NFA $\mathcal{A}$ which has a single final state.

(Basis:)

- Suppose $R = \emptyset$. Let $\mathcal{A}$ be

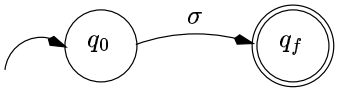

  Since there are no paths from the start state to the final state $L(\mathcal{A}) = \emptyset = L(\emptyset)$. Furthermore, $\mathcal{A}$ has a single final state.

- Suppose $R = \varepsilon$. Let $\mathcal{A}$ be



Here the empty string is in $L(\mathcal{A})$ since there is an $\varepsilon$-transition from the start state to the final state, but no other strings are in $L(\mathcal{A})$ since there are no other transitions out of either state. Again, $\mathcal{A}$ has a single final state.

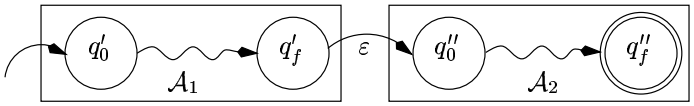
- Suppose $R = \sigma$ for some $\sigma \in \Sigma$. Let $\mathcal{A}$ be




Essentially the same argument applies in this case.

(Induction:)

- Suppose $R = S_1 \cdot S_2$ where $S_1$ and $S_2$ are regular expressions and that $L(S_1) = L(\mathcal{A}_1)$ and $(S_2) = L(\mathcal{A}_2)$ for NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ with single final states. Let $\mathcal{A}$ be




Formally, if $\mathcal{A}_1 = \langle Q_1, \Sigma, q'_0, \delta_1, \{q'_f\}\rangle$ and $\mathcal{A}_2 = \langle Q_2, \Sigma, q''_0, \delta_2, \{q''_f\}\rangle$ we let

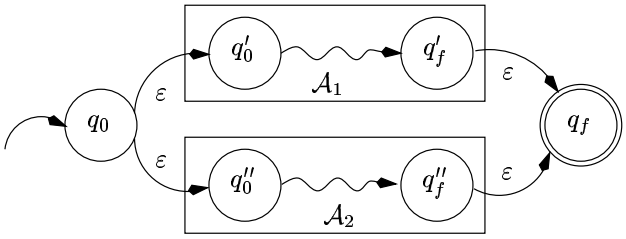
$$\mathcal{A} = \langle Q_1 \cup Q_2, \Sigma, q'_0, \delta, \{q''_f\}\rangle$$

where for all $q \in Q_1 \cup Q_2$ and $x \in \Sigma \cup \{\varepsilon\}$:

$$\delta(q,x) = \begin{cases} \delta_1(q,x) & \text{if } q \in Q_1 \setminus q'_f, \\ \delta_1(q'_f,x) & \text{if } q = q'_f \text{ and } x \neq \varepsilon, \\ \delta_1(q'_f,\varepsilon) \cup \{q''_0\} & \text{if } q = q'_f \text{ and } x = \varepsilon, \\ \delta_2(q,x) & \text{if } q \in Q_2. \end{cases}$$

Clearly there is a path from $q'_0$ to $q''_f$ labeled $w$ iff $w = u \cdot v$ where there is a path from $q'_0$ to $q'_f$ labeled $u$ and one from $q''_0$ to $q''_f$ labeled $v$. Thus $L(\mathcal{A}) = L(\mathcal{A}_1) \cdot L(\mathcal{A}_2) = L(S_1 \cdot S_2)$. Furthermore, $q'_f$ is the sole final state of $\mathcal{A}$.

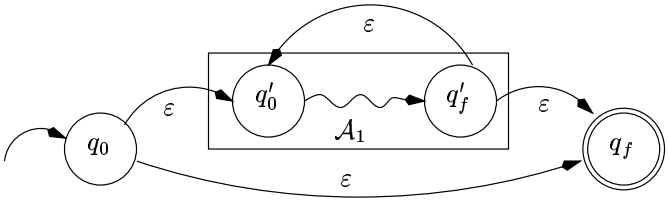
- Suppose $R = S_1 + S_2$ where $S_1$ and $S_2$ are regular expressions and that $L(S_1) = L(\mathcal{A}_1)$ and $L(S_2) = L(\mathcal{A}_2)$ for NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ with single final states. Let $\mathcal{A}$ be




30. Give the formal definition of $\mathcal{A}$ if $\mathcal{A}_1$ and $\mathcal{A}_2$ are as in the previous case.

Clearly there is a path from $q_0$ to $q_f$ labeled $w$ iff there is a path labeled $w$ from $q_0'$ to $q_f'$ or one from $q_0''$ to $q_f''$. Thus $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2) = L(S_1 + S_2)$.

- Suppose, finally, that $R = S_1^*$ and that $L(S_1) = L(\mathcal{A}_1)$, etc. Let $\mathcal{A}$ be




A similar formal construction applies. Here any path from $q_0$ to $q_f$ follows either the $\varepsilon$-transition directly from $q_0$ to $q_f$ or consists of the concatenation of one or more paths from $q_0'$ to $q_f'$.

31. Why does this construction not simply use the initial and final states of $\mathcal{A}_1$ as the initial and final states of $\mathcal{A}$, simply adding $\varepsilon$-transitions from $q_0'$ to $q_f'$ and from $q_f'$ to $q_0'$? Give an example of an automaton $\mathcal{A}_1$ for which the simpler construction fails. This example will *not* be an automaton which would be constructed from a regular expression. The simpler construction *will* work in the context of the proof, but the induction hypothesis needs to be strengthened slightly. How? Explain why this suffices.

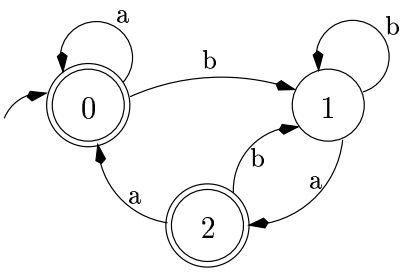
⊣

32. Construct an NFA accepting $L((a + bc)^*a + b)$.

## 10.2 Constructing Regular Expressions from DFAs

**Lemma 10** *If $L \subseteq \Sigma^*$ is $L(\mathcal{A})$ for a DFA $\mathcal{A}$ then it is $L(R)$ for a regular expression $R$.*

The approach we will use to proving the lemma involves constructing and solving a system of simultaneous equations defining, for each state, the set of strings labeling paths from the initial state to that state. For concreteness, we will work an example, based on the following DFA, while we develop the proof.




For each $q \in Q$ let

$$R_q \stackrel{\text{def}}{=} \{w \mid \hat{\delta}(q_0, w) = q\}.$$

Let's consider which strings are in $R_q$ for a given state $q$. Clearly $R_q$ includes $\varepsilon$ iff $q = q_0$. Otherwise, every string in $R_q$ is of the form $w\sigma$, where $w$ is in $R_p$ and there is a transition from state $p$ to state $q$ on $\sigma$ (i.e., $\delta(p, \sigma) = q$). Thus

$$R_q = \begin{cases} \{\varepsilon\} \cup \bigcup_{\delta(p,\sigma)=q} [R_p \cdot \{\sigma\}] & \text{if } q = q_0, \\ \bigcup_{\delta(p,\sigma)=q} [R_p \cdot \{\sigma\}] & \text{otherwise .} \end{cases}$$

For the example automaton, we get a system of three equations:

$$\begin{aligned} R_0 &= R_0 a + R_2 a + \varepsilon \\ R_1 &= R_0 b + R_1 b + R_2 b \\ R_2 &= R_1 a \end{aligned}$$

Here we have used, for instance, '$R_0 a$' to denote the concatenation of the languages $R_0$ and $\{a\}$ and '$+$' to denote union. (Equations of this form are known as *Regular Equations.*)

Our objective is to obtain a solution to the system of regular equations in the form of regular expressions denoting each of the $R_q$. Then, since the

language accepted by a DFA is the set of all strings labeling paths from the initial state to some final state, which is to say

$$L(\mathcal{A}) = \bigcup_{q \in F} [R_q],$$

we can obtain a regular expression denoting $L(\mathcal{A})$ by combining (with '+') the regular expressions denoting $R_q$ for each $q \in F$.

The question, then, is how to obtain the solution to the system of regular equations. The crucial step, of course, is eliminating occurrences of $R_q$ from the right hand side of the definition of $R_q$. For this we employ the following lemma.

**Lemma 11** *Let $P$, $Q$ and $R$ be sets of strings with $\varepsilon \notin P$. Then the equation*

$$R = Q + RP$$

*has a unique solution*

$$R = QP^*.$$

**Proof:** It is useful to consider what happens when we substitute $Q + RP$ into itself for $R$:

$$\begin{aligned} R &= Q + RP \\ &= Q + (Q + RP)P = Q + QP + RP^2 \\ &= Q + QP + (Q + RP)P^2 = Q + QP + QP^2 + RP^3 \\ &\vdots \\ &= Q + QP + QP^2 + \cdots + QP^i + RP^{i+1} \\ &\vdots \end{aligned}$$

Thus, $QP^i \subseteq R$ for all $i \geq 0$. Consequently, $\bigcup_{i \geq 0} [QP^i] \subseteq R$, which is to say, $QP^* \subseteq R$.

In fact, $R = QP^*$ is *a* solution to $R = Q + RP$, which we can verify by substituting $QP^*$ for $R$ in $R = Q + RP$:

$$QP^* = Q + QP^*P = Q(\varepsilon + PP^*) = QP^*.$$

We now need to show that this solution is unique. We will do this by showing that whenever $R = S$ is a solution then $S \subseteq QP^*$. Then, since $QP^* \subseteq R$ and $R = S$ we will have $S = QP^*$.
Suppose $R = S$ and $w \in S$. Then, from above, we have, for all $i \geq 0$,

$$w \in Q + QP + QP^2 + \cdots + QP^i + RP^{i+1},$$

and, in particular,

$$Q + QP + QP^2 + \cdots + QP^{|w|} + RP^{|w|+1}.$$

Now, since $\varepsilon \notin P$ every string in $RP^{|w|+1}$ is strictly longer than $|w|$. Thus $w \notin RP^{|w|+1}$. It must be the case, then, that

$$w \in Q + QP + QP^2 + \cdots + QP^i$$

and, therefore, $w \in QP^*$. Since the choice of $w$ was arbitrary, we have that $w \in S \Rightarrow w \in QP^*$. In other words, $S \subseteq QP^*$. ⊣

The proof of the uniqueness of $QP^*$ as a solution depends on the fact that $\varepsilon \notin P$ (otherwise $RP^{|w|+1}$ may include strings that are not strictly longer than $|w|$), but the proof that $QP^*$ is a solution does not. Presumably if $\varepsilon \in P$ then, while $QP^*$ will still be a solution, there may be other solutions.

33. Show that if $P = \{\varepsilon\}$ then every set that contains $Q$ as a subset is a solution to $R = Q + RP$.

34. Give an example of an $R$, $P$ and $Q$ such that $R = Q + RP$ but $R \neq QP^*$.

To see how to use this result to solve the system of equations, consider the equations we obtained for the example DFA. The idea is to focus on one equation at a time, identifying $Q$ and $P$ such that the equation is of the form $R = Q + RP$:

$$R_0 = R_0a + R_2a + \varepsilon = \underbrace{R_2a + \varepsilon}_{Q} + R_0 \underbrace{a}_{P}.$$

Then

$$\begin{aligned} R_0 &= QP^* \\ &= (R_2a + \varepsilon)a^* \\ &= R_2aa^* + a^*. \end{aligned}$$

Substituting $R_1a$ for $R_2$

$$R_0 = R_1aaa^* + a^*.$$

Substituting these into the equation for $R_1$

$$\begin{aligned} R_1 &= R_0b + R_1b + R_2b \\ &= (R_1aaa^* + a^*)b + R_1b + R_1ab \\ &= R_1aaa^*b + a^*b + R_1b + R_1ab \\ &= a^*b + R_1(b + ab + aaa^*b) \\ &= \underbrace{a^*b}_{Q} + R_1 \underbrace{a^*b}_{P}, \qquad \text{since } b + ab + aaa^*b = a^*b. \end{aligned}$$

Then

$$R_1 = a^*b(a^*b)^*.$$

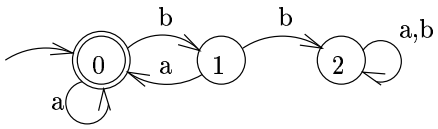
Finally, substituting this into the equations for $R_0$ and $R_2$:

$$\begin{aligned} R_0 &= a^*b(a^*b)^*aaa^* + a^* \text{ and} \\ R_2 &= a^*b(a^*b)^*a. \end{aligned}$$

It follows, then, that

$$\begin{aligned} L(\mathcal{A}) &= R_0 + R_2 \\ &= a^*b(a^*b)^*aaa^* + a^* + a^*b(a^*b)^*a \\ &= a^* + a^*b(a^*b)^*aa^*. \end{aligned}$$

35. Construct a regular expression denoting the language accepted by the following DFA.



# 11 The Pumping Lemma for Regular Languages

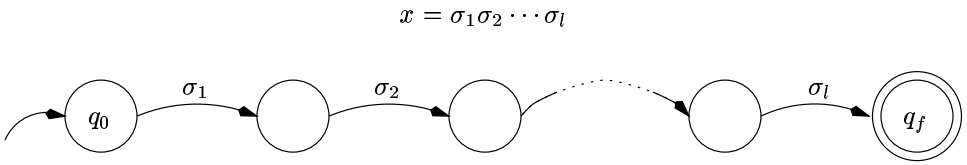
We know now how to prove that a language *is* regular; we have at least two ways—exhibit a DFA and prove that it accepts the language or exhibit a regular expression and prove that it denotes it. We know also that every regular language can be accepted by a DFA, which is to say that there is a finite bound (independent of the length of the input) on the amount of memory needed to recognized the strings in the language. But this is not a very powerful model. It seems likely that there are languages that cannot be recognized using a finitely bounded amount of memory. One example, from the informal examples we explored at the start of the tutorial, is the language

$$L_{ab} = \{a^i b^i \mid i \geq 0\}.$$

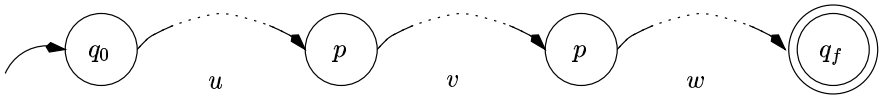
Here our intuition was that we can put no bound on number of '$a$'s we have to count in order to recognize the language. Consequently, it appears that this is not a regular language. The question is how to prove that a language is *not* regular.

Potentially, this is a much more difficult problem. To establish that a language is regular we need only exhibit a DFA that accepts it; to establish it is not regular we need to prove that it is not accepted by *any* DFA. Fortunately, by looking at the nature of DFAs we can identify properties that are characteristic of the languages they accept. Thus, the fact that regular languages are all accepted by DFAs implies that they share these properties. The approach we will use to prove non-regularity of a language is to show that it does not share at least one of these properties. Consequently, it cannot be accepted by a DFA. In this section, we will explore a closure property of the following sort: if a regular language includes strings of a certain type, then it includes all strings of a related type. We will be able to establish non-regularity of a given language, then, by exhibiting a string of the first type that is in the language along with a string of the related type that is not.

Lets start by considering what it means for a string to be accepted by a DFA. Suppose $\mathcal{A} = \langle Q, \Sigma, q_0, \delta, F\rangle$ is a DFA and that $Q = \{q_0, q_1, \ldots, q_{n-1}\}$ includes $n$ states. Thinking of the automaton in terms of its transition graph, a string $x$ is accepted by the automaton iff there is a path through the graph from $q_0$ to some $q_f \in F$ that is labeled $x$, i.e., if $\hat{\delta}(q_0, x) \in F$. Suppose $x \in L(\mathcal{A})$ and $|x| = l$. Then there is a path $l$ edges long from $q_0$ to $q_f$. Since the path traverses $l$ edges, it must visit $l + 1$ states.




Suppose, now, that $l \geq n$. Then the path must visit at least $n+1$ states. But there are only $n$ states in $Q$; thus, the path must visit at least one state at least twice. (This is an application of the *pigeon hole principle*: If one places $k$ objects into $n$ bins, where $k > n$, then at least one bin must contain at least two objects.)




Thus, whenever $|x| \geq n$ the path labeled $w$ will have a cycle. We can break the path into three segments: $x = uvw$, where

- there is a path (perhaps empty) from $q_0$ to $p$ labeled $u$ (i.e., $\hat{\delta}(q_0, u) = p$),
- there is a (non-empty) path from $p$ to $p$ (a cycle) labeled $v$ (i.e., $\hat{\delta}(p, v) = p$),
- there is a path (again, possibly empty) from $p$ to $q_f$ labeled $w$ (i.e., $\hat{\delta}(p, w) = q_f$).

But if there is a path from $q_0$ to $p$ labeled $u$ and one from $p$ to $q_f$ labeled $w$ then there is a path from $q_0$ to $q_f$ labeled $uw$ in which we do not take the loop labeled $v$, which is to say $uw \in L(\mathcal{A})$. Formally

$$\hat{\delta}(q_0, uw) = \hat{\delta}(\hat{\delta}(q_0, u), w) = \hat{\delta}(p, w) = q_f \in F.$$

Similarly, we can take the $v$ loop more than once:

$$\begin{aligned}
\hat{\delta}(q_0, uvvw) &= \hat{\delta}(\hat{\delta}(\hat{\delta}(\hat{\delta}(q_0, u), v), v), w) \\
&= \hat{\delta}(\hat{\delta}(\hat{\delta}(p, v), v), w) \\
&= \hat{\delta}(\hat{\delta}(p, v), w) \\
&= \hat{\delta}(p, w) = q_f \in F.
\end{aligned}$$

In fact, we can take it as many times as we like. Thus, $uv^iw \in L(\mathcal{A})$ for all $i$.

This implies, then, that if the language accepted by a DFA with $n$ states includes a string of length at least $n$ then it contains infinitely many closely related strings as well. We can strengthen this by noting (as a consequence of the pigeon hole principle again) that the length of the path from $q_0$ to the first time a state repeats (i.e., the second occurrence of $p$) must be less than $n$. Thus $|uv| \leq n$.

Now suppose $L$ is an arbitrary regular language. Then there is *some* DFA accepting it and that DFA has *some* fixed number of states. Thus there is a constant $n$ (the number of states in a DFA accepting $L$) such that if $L$ includes any string of length greater than or equal to $n$ then there is a non-empty segment of that string falling somewhere in its first $n$ positions that can be repeated (or *pumped*) any number of times (including zero) always producing strings in $L$. This result is known as the *Pumping Lemma* for regular languages.

**Lemma 12 (Pumping Lemma)** *For every regular language $L$ there is a constant $n$ depending only on $L$ such that, for all strings $x \in L$ if $|x| \geq n$ then there are strings $u$, $v$ and $w$ such that*

1. $x = uvw$,
2. $|uv| \leq n$,
3. $|v| \geq 1$,
4. *for all* $i \geq 0$, $\quad uv^i w \in L$.

What this says is that if there is any string in $L$ "long enough" then there is *some* family of strings of related from that are all in $L$, that is, that there is *some* way of breaking the string into segments $uvw$ for which $uv^i w$ is in $L$ for all $i$. It *does not* say that *every* family of strings of related form is in $L$, that $uv^i w$ will be in $L$ for every way of breaking the string into three segments $uvw$. It also *does not* say that every long string in $L$ is of form $uv^i w$ for some $i > 1$ (i.e., it *does not* say that every long string in $L$ has some part that repeats). It does not, in fact, say anything about individual strings at all, it simply lets us identify families of strings all of which must be in $L$. The way we will use this is to identify such a family that should be in $L$ if $L$ is regular for which we can show at least one string in the family is not in $L$.

Note that this lemma talks *only* about the strings in the language. While we justified it by appealing to the fact that a language is regular iff there is a

DFA that accepts it, we don't need to actually come up with such a DFA to apply it. On the other hand, if we actually *have* a DFA for the language we can fix a concrete upper bound on the constant of the lemma—$n$ is no larger than the number of states in $Q$. Thus we can strengthen the pumping lemma slightly, by adding:

**Lemma 13** *Suppose $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ is a DFA witnessing the fact that $L$ is regular. Then the constant $n$ of the pumping lemma is no greater than* $\mathbf{card}(Q)$.

## 11.1 Applying the Pumping Lemma

To establish that a language $L$ is not regular using the pumping lemma we need to show that the pumping lemma is not true for that language, i.e., for *any* $n$ there is *some* $x \in L$ with $|x| \geq n$ such that for *all* $u$, $v$ and $w$ where $uvw = x$, $\quad |uv| \leq n$ and $|v| \geq 1$ there is *some* $i$ for which $uv^iw \notin L$.

A useful way to think of this is as a game between you, who are trying to prove that the pumping lemma fails for $L$, and an adversary that is trying to prove that it holds. Expressed formally, the lemma says ('$\forall$' and '$\exists$' should be read "for all" and "there exists", respectively):

$$
\begin{array}{l}
(\forall L)[L \text{ regular } \Rightarrow \\
\quad (\exists n)[ \\
\quad\quad (\forall x)[x \in L \text{ and } |x| \geq n \Rightarrow \\
\quad\quad\quad (\exists u, v, w)[x = uvw \text{ and} \\
\quad\quad\quad\quad\quad\quad |uv| \leq n \text{ and} \\
\quad\quad\quad\quad\quad\quad |v| \geq 1 \text{ and} \\
\quad\quad\quad\quad\quad\quad (\forall i \geq 0)[uv^iw \in L]]]]].
\end{array}
$$

Every '$\forall$' is your choice—the lemma should be true for any value you choose. Every '$\exists$' is your opponent's choice—they need only exhibit some value for which the lemma is true. Each move can depend on all the choices made prior to it. The game starts with your choice of the $L$ you wish to prove to be non-regular. You opponent then chooses some $n$, you choose a string $x \in L$ of length at least $n$, etc. You win if, at the end of this process, you can choose $i$ such that $uv^iw \notin L$. To establish that the lemma is not satisfied by $L$ you have to show that no matter which choices your adversary makes, you can always have a winning choice of $x$ and $i$, that is, you must give a strategy, accounting for all possible choices of your opponent, that always leads to a win for you.

Of course, your strategy at each step will depend on the choices your opponent has made.

What we end up with is a proof by contradiction. For instance:

To show that $L_{ab} = \{a^j b^j \mid j \geq 0\}$ is not regular.

**Proof:** Suppose, by way of contradiction, $L_{ab}$ is regular. (Your choice of $L$.)
Let $n$ be the constant of the pumping lemma. (Your adversary chooses $n$. You can make no assumptions about $n$ but you will base your subsequent choices on its value.)
Let $x = a^n b^n$. (Your choice of $x$. Note that it depends on $n$. In fact, if your choice does not depend $n$, if $n$ is not a parameter of your definition of $x$, then the proof will almost certainly fail.)
Since $x \in L_{ab}$ and $|x| \geq n$, by the pumping lemma there must be some $u$, $v$ and $w$ such that $x = uvw$, $|uv| \leq n$, $|v| \geq 1$ and $uv^i w \in L_{ab}$ for all $i$. (You opponent chooses how to split $x$ into $uvw$ subject only to the conditions that $|uv| \leq n$ and $|v| \geq 1$. Your strategy must account for all ways of doing this—all you can assume about $u$, $v$ and $w$ is that they meet the conditions of the lemma.)
Since $|uv| \leq n$, both $u$ and $v$ must fall within the first $n$ symbols of $x$. Thus, $u$ and $v$ must consist only of '$a$'s. Furthermore, $v$ must include at least one '$a$', since $|v| \geq 1$. Thus, there must be some $j$, $k$ and $l$ such that

$$u = a^j, \qquad v = a^k, \qquad w = a^l b^n, \qquad j + k + l = n, \text{ and } k \geq 1.$$

But suppose $i = 0$. (Your choice of $i$.) Then

$$uv^0 w = uw = a^j a^l b^n$$

where $j + l = n - k$ which is strictly less than $n$. Thus $uv^i w \notin L_{ab}$ for $i = 0$ and the pumping lemma does not hold for $L_{ab}$, contradicting our assumption that $L_{ab}$ was regular. ⊣

In this case our choice of $x$ restricted the adversary's choice of $uvw$ enough that all choices can be treated with a single argument. In general this will not be the case and, even if we can use the same $i$ in all choices, we will have to account separately for each alternative way of dividing $x$.

**Example:** Let $L_4 \subseteq \{a,b\}^*\{c,d\}^*$ be the language in which a string $w$ is in $L_4$ iff the number of occurrences of the substring $ab$ in $w$ is greater than or equal to the number of occurrences of the substring $cd$ in $w$. Prove that $L_4$ is non-regular using the pumping lemma.

**Proof:** Suppose, for contradiction, that $L_4$ is regular. Let $n$ be the constant of the pumping lemma. Let $x = (ab)^n(cd)^n$. Let $|w|_{ab}$ denote the number of occurrences of the substring $ab$ in $w$. Similarly for $|w|_{cd}$. Then $x \in L_4$, $|x| > n$, and $|x|_{ab} = |x|_{cd} = n$.

By the pumping lemma, $x = uvw$, where $|uv| \leq n$, $|v| \geq 1$ and $uw \in L_4$. Consequently, $uv$ is a prefix of $(ab)^n$.
We must now account for every way in which a prefix of $(ab)^n$ can be divided into $uv$ with $|v| \geq 1$. Any way of dividing these possibilities into cases is acceptable, *as long as it is exhaustive.* One way that works well for this language is to note that $v$ must start with either an '$a$' or a '$b$' and, similarly, must end with one or the other. Thus, there are four cases:

$$v = a(ba)^j \qquad v = a(ba)^j b \qquad v = b(ab)^j a \text{ or } v = b(ab)^j, \qquad j \geq 0.$$

By cases, then:

$$\begin{array}{llll} u = (ab)^i & v = a(ba)^j & w = b(ab)^k(cd)^n & \Rightarrow \quad |uw|_{ab} = n - 1 - |v|_{ab} \\ u = (ab)^i & v = a(ba)^j b & w = (ab)^k(cd)^n & \Rightarrow \quad |uw|_{ab} = n - |v|_{ab} \text{ and } |v|_{ab} > 0. \\ u = (ab)^i a & v = b(ab)^j a & w = b(ab)^k(cd)^n & \Rightarrow \quad |uw|_{ab} = n - 1 - |v|_{ab} \\ u = (ab)^i a & v = b(ab)^j & w = (ab)^k(cd)^n & \Rightarrow \quad |uw|_{ab} = n - 1 - |v|_{ab} \end{array}$$

Thus in all cases $|uw|_{ab} < |uw|_{cd} = n$, $uw \notin L_4$ and $L_4$ is not regular. ⊣

36. Prove that $L_5 \stackrel{\text{def}}{=} \{w \in \{a,b\}^* \mid |w|_a < |w|_b\}$ is not regular (where $|w|_a$ is the number of '$a$'s occurring in $w$).

37. Prove that the following language $L_6$ is not regular.

$$L_6 \stackrel{\text{def}}{=} \{w \in \{a,b\}^* \mid |w|_a < |w|_b \text{ if } |w|_a \text{ even, } |w|_a > |w|_b \text{ otherwise } \}$$

# 12 The Myhill-Nerode Theorem

As we have defined them, in every DFA there is a path from the start state to some state of the DFA for every string in $\Sigma^*$. Since there are infinitely many strings but only finitely many states many strings must lead to the same state. Recall that the state represents all the information that the DFA has about the string leading to that state, so, in a strong sense it is the nature of DFAs that they are *forgetful*—they are unable to remember a complete description of the strings they scan and must, inevitably, fail to distinguish some strings. The content of the Myhill-Nerode Theorem, the topic of this section, is that this partitioning of $\Sigma^*$ into finitely many classes of indistinguishable strings is characteristic of the regular languages.

Suppose $L$ is regular. Then there is some DFA $\mathcal{A}$ that accepts it, $L = L(\mathcal{A})$. The approach we have taken to proving that a DFA accepts a language is to identify each state with an invariant, a property that is characteristic of the strings that label paths from the start state to that state. These invariants define a relation on strings: two strings are related iff they satisfy the same invariant, which is to say iff they lead to the same state. We will refer to this relation as $R_{\mathcal{A}}$:

$$R_{\mathcal{A}} \stackrel{\text{def}}{=} \{\langle w, v\rangle \mid \hat{\delta}(q_0, w) = \hat{\delta}(q_0, v)\}$$

and will use $R_{\mathcal{A}}$ as an 'infix' relation symbol:

$$wR_{\mathcal{A}}v \Leftrightarrow \langle w, v\rangle \in R_{\mathcal{A}}.$$

Two strings are equivalent, from the point of view of $\mathcal{A}$, iff they are related by $R_{\mathcal{A}}$. We can verify that $R_{\mathcal{A}}$ is an *equivalence relation*—it is *reflexive*: $wR_{\mathcal{A}}w$ for all $w$; it is *symmetric*: if $wR_{\mathcal{A}}v$ then $vR_{\mathcal{A}}w$ as well; and it is *transitive*: $uR_{\mathcal{A}}v$ and $vR_{\mathcal{A}}w$ implies $uR_{\mathcal{A}}w$; so it is consistent with this interpretation.

Let

$$[w]_{R_{\mathcal{A}}} \stackrel{\text{def}}{=} \{v \mid wR_{\mathcal{A}}v\}.$$

This is the *equivalence class of $w$ wrt $R_{\mathcal{A}}$*, the set of all strings equivalent to $w$ in the $R_{\mathcal{A}}$ sense. Note that, as with all equivalence classes, every string $w \in \Sigma^*$ is in some class (namely $[w]_{R_{\mathcal{A}}}$) and no string is in more than one ($w \in [u]_{R_{\mathcal{A}}}$ and $w \in [v]_{R_{\mathcal{A}}}$ implies $[u]_{R_{\mathcal{A}}} = [v]_{R_{\mathcal{A}}}$). Thus the equivalence classes of $R_{\mathcal{A}}$ *partition* $\Sigma^*$—they are disjoint and their union is $\Sigma^*$.

Now, we potentially have many names for each equivalence class since if $wR_{\mathcal{A}}v$ then $[w]_{R_{\mathcal{A}}} = [v]_{R_{\mathcal{A}}}$. We can capture the set of all classes by choosing an

(arbitrary) canonical representative for each class. This set of representatives is referred to as an *index set* or *spanning set* (we will denote it with $I$) and

$$\Sigma^* = \bigcup_{w \in I} [[w]_{R_\mathcal{A}}] .$$

Given that $R_\mathcal{A}$ is defined by the automaton $\mathcal{A}$, how big is its index set? Certainly, there can be no more equivalence classes than there are states of $\mathcal{A}$ (there can be fewer, since some states of $\mathcal{A}$ may be inaccessible from $q_0$). Thus the number of equivalence classes wrt $R_\mathcal{A}$ is finite; we say that $R_\mathcal{A}$ has *finite index* (or is spanned by a finite set).

Since the finiteness of the number of equivalence classes of $R_\mathcal{A}$ is a consequence of the key characteristic of DFAs—the finiteness of their state sets—it should seem plausible that the existence of *some* relation $R_\mathcal{A}$ with finite index is a key property of regular sets. To pin this down we need to look a little more closely at the nature of $R_\mathcal{A}$ and its relationship to $L$.

First, note that, since two strings $w$ and $u$ are related by $R_\mathcal{A}$ iff the paths (from the start state) they label lead to the same state, it must be the case that if we extend them both with the same string we will obtain, again, equivalent strings, i.e.,

$$wR_\mathcal{A}u \Rightarrow (\forall v)[wvR_\mathcal{A}uv].$$

We say that $R_\mathcal{A}$ is *right invariant* wrt concatenation.

Finally, we note that

$$L = \bigcup_{\hat{\delta}(q_0,w) \in F} [[w]_{R_\mathcal{A}}] ,$$

which is to say, $L$ is the union of some of the equivalence classes of $\Sigma^*$ wrt $R_\mathcal{A}$.

When we put these all together, we get a lemma.

**Lemma 14** *If a language $L \subseteq \Sigma^*$ is regular then it is the union of some of the equivalence classes of a right invariant equivalence relation on $\Sigma^*$ which is of finite index.*

Note that, while given $\mathcal{A}$ accepting $L$ we can immediately produce $R_\mathcal{A}$, the lemma is actually independent of $\mathcal{A}$. If we know that $L$ is regular then we know that some relation with the properties we have established for $R_\mathcal{A}$ exists, since some $\mathcal{A}$ does. Intuitively, the converse ought to be true as well—if such an equivalence relation exists then we ought to be able to construct a DFA using

the equivalence classes of the relation as states; the existence of such a relation ought to imply that $L$ is regular. And, in fact, it does but it is useful to prove this in a somewhat round-about way.

The fact that $R_{\mathcal{A}}$ is right invariant wrt concatenation implies that

$$wR_{\mathcal{A}}u \Rightarrow (\forall v)[wv \in L \Leftrightarrow uv \in L].$$

Let's consider the relation

$$R_L \stackrel{\text{def}}{=} \{\langle w, u\rangle \mid (\forall v)[wv \in L \Leftrightarrow uv \in L]\}.$$

Note that, although this is also an equivalence relation, it is *not* the same relation as $R_{\mathcal{A}}$ because it may be the case that $\mathcal{A}$ distinguishes two strings $w$ and $u$ even though $wR_Lu$—$w$ and $u$ may lead to different states of $\mathcal{A}$, even though $\mathcal{A}$ behaves the same (in the sense of accepting or not) on all strings extending them. However, as we just saw, if $wR_{\mathcal{A}}u$ then $wR_Lu$, thus every equivalence class wrt $R_{\mathcal{A}}$ is a subset of an equivalence class wrt $R_L$; the equivalence classes of $R_{\mathcal{A}}$ do not *break* the equivalence classes of $R_L$, rather they partition them. We say that $R_{\mathcal{A}}$ is a *refinement* of $R_L$—every equivalence class wrt $R_L$ is the union of the equivalence classes of $R_{\mathcal{A}}$ it intersects. Thus, the number of equivalence classes of $R_L$ can be no greater than the number of equivalence classes of $R_{\mathcal{A}}$. Consequently, the number of equivalence classes of $\Sigma^*$ wrt $R_L$ is also finite. $R_L$ has finite index. This leads to our second lemma.

**Lemma 15** *Suppose* $L \subseteq \Sigma^*$. *Let* $R_L \stackrel{\text{def}}{=} \{\langle w, u\rangle \mid (\forall v)[wv \in L \Leftrightarrow uv \in L]\}$. *If* $L$ *is regular then* $R_L$ *has finite index.*

This follows from the previous lemma and the fact that right invariance of $R_{\mathcal{A}}$ implies that $R_{\mathcal{A}}$ refines $R_L$.

We are close, now, to having characterizations of the regular languages in terms of both $R_{\mathcal{A}}$ and $R_L$. If we can show that whenever $R_L$ has finite index then $L$ is regular we will get not only that this characterizes the regular sets but also that being the union of some of the classes of a right invariant equivalence relation of finite index does, since it is implied by regularity and implies finiteness of index of $R_L$.

To obtain this result, we will follow the idea we sketched for $R_{\mathcal{A}}$—we will construct a DFA accepting $L$ using the equivalence classes of $\Sigma^*$ wrt $R_L$ as our state set.

Let $Q' = \{[w]_{R_L} \mid w \in \Sigma^*\}$. (Note that while each of these states is a potentially infinite set, the number of states is finite. To avoid even this

vestigial infiniteness, we could take $Q'$ to be any index set for $R_L$, but this definition is slightly simpler.)

Let $\delta'([w]_{R_L}, \sigma) = [w\sigma]_{R_L}$.

We need to verify that this is, in fact, well defined. In particular, we need to show that for all $u$ and $\sigma$, if $u \in [w]_{R_L}$ then $u\sigma \in [w\sigma]_{R_L}$. (In other words, we need to show that $R_L$ is right invariant, but this does not follow from right invariance of $R_{\mathcal{A}}$ since $uR_Lw$ does not imply $uR_{\mathcal{A}}w$.) To see that this is the case, suppose $uR_Lw$. Then, for all $v$, $uv \in L \Leftrightarrow wv \in L$. And, in particular (letting $v = \sigma v'$), for all $v'$, $u\sigma v' \in L \Leftrightarrow w\sigma v' \in L$. Thus, $u\sigma R_L w\sigma$.

Finally, let $q'_0 = [\varepsilon]_{R_L}$ and $F' = \{[w]_{R_L} \mid w \in L\}$.

**Lemma 16** *Let $\mathcal{A}' = \langle Q', \Sigma, q'_0, \delta', F' \rangle$, where $Q'$, $q'_0$, $\delta'$ and $F'$ are as described above. Then $L = L(\mathcal{A}')$.*

38. Prove that, for all $w \in \Sigma^*$, $\hat{\delta}'(q'_0, w) = [w]_{R_L}$.

39. Use this to prove the lemma.

Putting the three lemmas together we get:

**Theorem 2 (Myhill-Nerode)** *The following are equivalent:*

- *$L$ is regular.*
- *$L$ is the union of some of the equivalence classes of a right invariant equivalence relation of finite index.*
- *The relation $R_L \stackrel{\text{def}}{=} \{\langle w, u\rangle \mid (\forall v)[wv \in L \Leftrightarrow uv \in L]\}$ has finite index.*

Note that with $R_L$ we have abandoned the notion of DFA entirely. This is a relation that is defined directly in terms of the strings that are and are not in the language, independent of any particular mechanism for defining or accepting it.

## 12.1 Using Myhill-Nerode to Prove Non-Regularity

The fact that a language $L$ is regular iff $R_L$ has finite index gives us another approach to proving languages are not regular. If we can show that there must be infinitely many equivalence classes of $\Sigma^*$ wrt $R_L$ then $L$ cannot be regular. This is often simpler than using the Pumping Lemma.

The most direct approach to doing this is to give an infinite sequence of strings in $\Sigma^*$ all of which are distinct wrt to $R_L$. Since they each must be in a distinct equivalence class and since there are infinitely many of them, $R_L$ cannot have finite index.

**Example:** To show that

$$L_{ab} = \{a^i b^i \mid i \geq 0\}.$$

is not regular, consider the sequence of strings

$$\langle a^0, a^2, \ldots, a^i, \ldots \mid i \geq 0 \rangle.$$

We claim that every string in this sequence is distinct wrt $R_L$ from every other string in the sequence. To see this, consider $a^i$ and $a^j$ for $i \neq j$. Then $a^i b^i \in L_{ab}$ while $a^j b^i \notin L_{ab}$. Thus, $b^i$ witnesses that $a^i$ and $a^j$ are *not* related by $R_L$. Since $i$ and $j$ are arbitrary, every string in the sequence is distinct, wrt $R_L$, from every other string in the sequence; no two share the same equivalence class. It follows that there must be at least as many equivalence classes of $R_L$ as there are strings in the sequence; $R_L$ cannot have finite index.

40. Consider, again, the problem of specifying schedules for a machine tool (see Section 8.3). Suppose, in this instance, that there is just a single type of part which requires two operations $A_1$ and $A_2$ to complete. These can be done in any order, although we will assume that operations always complete partially completed parts if they can. Thus, at any given time the partially completed parts will all be waiting for the same operation (although which operation can change over time). Assume, further, that there is an unbounded amount of space to store partially completed parts; the only constraint on a schedule is that every part gets completed.

    (a) Describe this language.

    (b) Use Myhill-Nerode to show that the set of feasible schedules, given these constraints, is not regular.

## 12.2 Using Myhill-Nerode to Prove Regularity

The Myhill-Nerode theorem gives us a characterization of the regular languages—a language is regular *iff* the second and third part of theorem are satisfied.

Thus, in contrast to the Pumping Lemma, we can use Myhill-Nerode not only to prove languages are not regular, but also to prove that languages *are* regular. In fact, one of the attractive features of this approach is the fact that if one's attempt to prove a language *is not* regular fails then one is likely to be well along the way to proving that the language *is* regular.

While it is possible to use either $R_{\mathcal{A}}$ or $R_L$ to prove a language is regular, using $R_L$ is usually much simpler. The idea is to consider the way in which $\Sigma^*$ is partitioned by $R_L$ and argue that there are only finitely many partitions.

**Example:** Consider, again, the example of Section 8.3. Let $L_2$ be the set of feasible schedules under the constraints given there. Following the Myhill-Nerode Theorem, let

$$wR_{L_2}u \stackrel{\text{def}}{\Longleftrightarrow} (\forall v)[wv \in L_2 \Leftrightarrow uv \in L_2].$$

Now, for any strings $w, v \in \{A_1, A_2, B_1, B_2\}^*$, it will be the case that $wv \in L_2$ iff

- *$v$ completes $w$*:

$$|w|_{A_1} - |w|_{A_2} = |v|_{A_2} - |v|_{A_1} \text{ and } |w|_{B_1} - |w|_{B_2} = |v|_{B_2} - |v|_{B_1}.$$

- *$wv$ does not satisfy*:

$$\text{FAIL}(x) \stackrel{\text{def}}{\Longleftrightarrow} x = x'x'' \text{ where } \left||x'|_{A_1} - |x'|_{A_2}\right| + \left||x'|_{B_1} - |x'|_{B_2}\right| > 2,$$

The first condition says that every part pending at the end of the schedule $w$ will be completed during the schedule $v$. The second says that there will never be more than two pending parts during this process.

Consider, now, an arbitrary strings in $w, u \in \{A_1, A_2, B_1, B_2\}^*$. Under what circumstances will it be the case that $wR_{L_2}u$? Well, if FAIL($w$) is true, then there will be no $v$ for which $wv \in L_2$. On the other hand, if FAIL($u$) is not true, then there is at least one $v$ for which $uv \in L_2$—the one that simply completes the parts outstanding at the end of $u$. Thus, if FAIL($w$) and FAIL($u$) then $wR_{L_2}u$, but if FAIL($w$) and not FAIL($u$) or *vice versa* then *not* $wR_{L_2}u$. It follows that the class of all strings that satisfy FAIL is one of the equivalence classes of $\{A_1, A_2, B_1, B_2\}^*$ wrt $R_{L_2}$. (This should not be a surprise, the class corresponds to the state Fail of the DFA we constructed for this language.)

It remains to consider what determines if two strings neither of which satisfy FAIL are related by $R_{L_2}$. First of all, it should be clear that a schedule $v$ will complete both $w$ and $u$ iff the set of parts outstanding at the end of $w$ is the same as the set outstanding at the end of $u$. For example,

$$(|w|_{A_1} - |w|_{A_2} = |v|_{A_2} - |v|_{A_1} \text{ and } |u|_{A_1} - |u|_{A_2} = |v|_{A_2} - |v|_{A_1}) \Leftrightarrow |w|_{A_1} - |w|_{A_2} = |u|_{A_2} - |u|_{A_1}.$$

Moreover, if the set of parts outstanding at the end of $w$ is the same as the set outstanding at the end of $u$ then, for all $v$, $wv$ will satisfy FAIL iff $uv$ satisfies FAIL. Conversely, it is not hard to see that if the set of parts outstanding at the end of $w$ and $u$ are *not* the same then there will be a schedule $v$ which completes one while FAILing on the other.

Thus, for $w, u \in \{A_1, A_2, B_1, B_2\}^*$, $wR_{L_2}u$ iff

- FAIL($w$) and FAIL($u$), or
- Not FAIL($w$) and not FAIL($u$) and $|w|_{A_1} - |w|_{A_2} = |u|_{A_2} - |u|_{A_1}$ and $|w|_{B_1} - |w|_{B_2} = |u|_{B_2} - |u|_{B_1}$.

It remains, to count how many equivalence classes this relation generates. There will be one for strings that satisfy fail plus one for each pair of values of $|w|_{A_1} - |w|_{A_2}$ and $|w|_{B_1} - |w|_{B_2}$ for $w$ that do not satisfy FAIL. Clearly, since $w$ does not satisfy FAIL,

$$-2 \leq |w|_{A_1} - |w|_{A_2}, |w|_{B_1} - |w|_{B_2} \leq 2.$$

It follows that there can be no more than $5 \cdot 5 = 25$ such pairs and $25 + 1 = 26$ classes altogether. (There are actually just 14.) Thus, $R_{L_2}$ has finite index and, by the Myhill-Nerode Theorem, $L_2$ is regular.

41. Using the Myhill-Nerode Theorem, prove the language $L_3$ of Problem 25 is regular.

## 12.3 Minimization of DFAs

Recall that we showed above that $R_{\mathcal{A}}$ was a refinement of $R_L$; if two strings are related by $R_{\mathcal{A}}$ they are necessarily related by $R_L$. The converse of this is not true—it may be the case that $\mathcal{A}$ distinguishes two strings, that the paths from the initial state labeled with the strings lead to distinct states in $\mathcal{A}$, even though any string labeling a path to some final state from one of those states

also labels a path to some final state from the other and *vice versa*. But, certainly, there is no need to distinguish these states. Since the behavior of the automaton is the same from both states, it should be possible to merge them into a single state. (This is not as immediate as it seems. The fact that a path labeled with some string leads to a final state from the one if and only if it leads to some final state from the other does not imply that they lead to the *same* final state. We may, in general, have to merge a number of states in order to preserve a deterministic transition function.)

Thus, if there is more than a single equivalence class wrt $R_{\mathcal{A}}$ partitioning any equivalence class wrt $R_L$ the automaton $\mathcal{A}$ includes more states than necessary; there is a simpler automaton that accepts the same language. Consider, now, whether any of the classes wrt $R_L$ are redundant. These are, after all, the classes we chose as the state set of the automaton we constructed in the proof of the Myhill-Nerode Theorem. (For that DFA, in fact, $R_{\mathcal{A}} = R_L$.) Is it possible to merge any of these states, to construct a simpler automaton that accepts the same language?

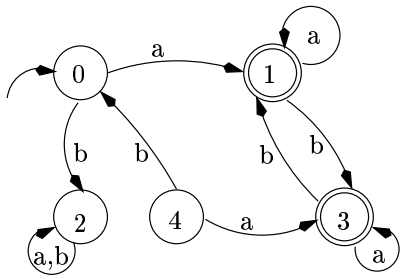
Suppose $[w]_{R_L} \neq [u]_{R_L}$. Then, by the definition of $R_L$ there must be some string $v$ such that $wv \in L$ while $uv \notin L$ or *vice versa*. But if we were to merge these two states the automaton would have to either accept both $wv$ and $uv$ or reject them both. Thus, if $[w]_{R_L} \neq [u]_{R_L}$ then every DFA that accepts $L$ will need to distinguish them. It follows that the DFA we constructed on the equivalence classes wrt $R_L$ is, in fact, *minimal*—there is no DFA with fewer states that accepts $L$. Moreover, while there will be other DFAs with the same number of states that accept $L$ (since we could take any finite set of the same size to be $Q$), every one of these will have to distinguish exactly the same sets of strings; necessarily $R_{\mathcal{A}} = R_L$ for all of them. It follows, then, that the only distinction between the minimal DFAs accepting $L$ is the labeling of the states. We say that they are *isomorphic*.

**Lemma 17** *The DFA constructed on $R_L$ is minimal in the size of its state set among DFAs accepting $L$. Moreover, up to isomorphism, it is the* unique *minimal DFA accepting $L$.*

This gives us a technique for minimizing DFAs, for eliminating redundant states. As DFAs employed in applications can get quite large, such minimalization can have a significant effect on efficiency. The idea is to identify classes of $R_{\mathcal{A}}$, states of $\mathcal{A}$, that are *indistinguishable* wrt $R_L$, where a pair of states $q, p \in Q$ are *distinguished* wrt $R_L$ iff there is a string $v$ which leads to a final

state from one and to a non-final state from the other, i.e., iff $\delta(q, v) \in F$ and $\delta(p, v) \notin F$, or *vice versa*. As we noted above, whether such a string exists is not necessarily obvious. The length of the string $v$ could, potentially, be quite long. The approach we will take to identifying pairs of states that are distinguished wrt $R_L$ is to iterate through $i \geq 0$ identifying all those pairs that are distinguished by strings of length $i$ and then using those to identify those pairs distinguished by strings of length $i + 1$, etc. This leads to a dynamic programming algorithm, which we will lay out by example.

Let $\mathcal{A}$ be the DFA:




We will construct a table relating pairs of states in which the entry in the $q$ row and the $p$ column will be non-empty iff we have distinguished states $q$ and $p$. Since the relation of being distinguished is symmetric, we need only the lower triangular of this table.

| | | | | |
|---|---|---|---|---|
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| | 0 | 1 | 2 | 3 |

In the first iteration $i = 0$. We will mark each pair of states that are distinguished by a string of length 0, which is to say we will mark the entry for $\langle p, q \rangle$ iff $\hat{\delta}(p, \varepsilon) \in F$ and $\hat{\delta}(q, \varepsilon) \notin F$ or *vice versa*. In other words, we distinguish every state in $F$ from every state not in $F$.

| | | | | |
|---|---|---|---|---|
| 1 | $\varepsilon$ | | | |
| 2 | | $\varepsilon$ | | |
| 3 | $\varepsilon$ | | $\varepsilon$ | |
| 4 | | $\varepsilon$ | | $\varepsilon$ |
| | 0 | 1 | 2 | 3 |

(We have marked them with the string that distinguishes them.)

In the subsequent iterations we identify states that are distinguished by increasingly long strings. A pair of states $\langle p, q\rangle$ will be distinguished by a string of length $i+1$ iff there is some $\sigma \in \Sigma$ for which the state reached from $p$ on $\sigma$ and the state reached from $q$ on $\sigma$ are distinguished by a path of length $i$, i.e., if $\delta(p,\sigma)$ and $\delta(q,\sigma)$ were distinguished in iteration $i$. Thus, in each iteration, we work our way through the table marking each entry $\langle p, q\rangle$ for which the entry $\langle \delta(p,\sigma), \delta(q,\sigma)\rangle$ (or *vice versa*) is already marked.

| | | | | |
|---|---|---|---|---|
| 1 | $\varepsilon$ | | | |
| 2 | a | $\varepsilon$ | | |
| 3 | $\varepsilon$ | | $\varepsilon$ | |
| 4 | | $\varepsilon$ | a | $\varepsilon$ |
| | 0 | 1 | 2 | 3 |

$\delta(0,a) = 1 \quad \delta(2,a) = 2$
$\delta(2,a) = 2 \quad \delta(4,a) = 3$

| | | | | |
|---|---|---|---|---|
| 1 | $\varepsilon$ | | | |
| 2 | a | $\varepsilon$ | | |
| 3 | $\varepsilon$ | | $\varepsilon$ | |
| 4 | ba | $\varepsilon$ | a | $\varepsilon$ |
| | 0 | 1 | 2 | 3 |

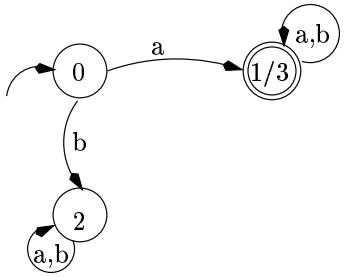
$\delta(0,b) = 2 \quad \delta(4,b) = 0$

We repeat this until some iteration fails to distinguish any new pairs of states. It should be clear that from that point on no more pairs will be distinguished. That all distinguishable pairs will have been marked at that point follows from the invariant:

> An entry $\langle p, q\rangle$ will be marked in the table at iteration $i$ iff there is a string $v$ of length no greater than $i$ for which $\hat{\delta}(p, v) \in F$ while $\hat{\delta}(q, v) \notin F$ or *vice versa*

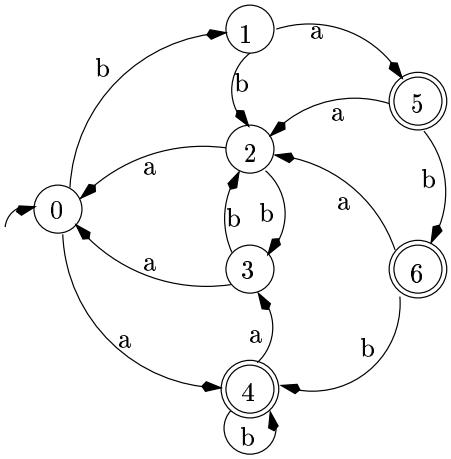
which can be easily established by induction on $i$. That the algorithm always terminates follows from the fact that the table is finite—one cannot distinguish any more pairs than there are entries in the table, thus, the algorithm converges after no more than that many iterations.

In the example, the pair $\langle 3, 1\rangle$ is left unmarked, thus, these states are indistinguishable wrt $R_L$ and can be merged. Note, also, that, while state 4 is distinguished from every other state it is unreachable from the initial state and is, therefore, useless. We can simply eliminate it.




Note that, in merging state $p$ and $q$ the new transition function will be well defined iff $\delta(p, \sigma)$ and $\delta(q, \sigma)$ are equivalent states. The algorithm guarantees this will be the case (why?).

42. Minimize the following DFA.



# 13 Closure Properties of the Class of Regular Languages

The pumping lemma gives a kind of closure property for individual regular languages: if the language includes a string of a particular form then it includes all strings of a related form. In this section we will look at some of the closure properties of the *class* of regular languages—properties of the form: if $L_1, \ldots, L_k$ are in that class (are regular) then the languages formed from the $L_i$ by some particular operation are all in the class as well.

## 13.1 Boolean Operations

**Theorem 3** *The class of regular languages is closed under union, concatenation, and Kleene closure.*

**Proof:** This follows immediately from the definition of regular languages. $\dashv$
Note that we actually gave constructive proofs of these closure results as part of the proof that every regular language is accepted by some DFA.

**Theorem 4** *The class of regular languages is closed under intersection.*

**Proof** (sketch)**:** Suppose $L_1$ and $L_2$ are regular. Then there are DFAs $\mathcal{M}_1 = \langle Q, \Sigma, \delta_1, q_0, F_1 \rangle$ and $\mathcal{M}_2 = \langle P, \Sigma, \delta_2, p_0, F_2 \rangle$ such that $L_1 = L(\mathcal{M}_1)$ and $L_2 = L(\mathcal{M}_2)$. We construct $\mathcal{M}'$ such that $L(\mathcal{M}') = L_1 \cap L_2$. The idea is to have $\mathcal{M}'$ run $\mathcal{M}_1$ and $\mathcal{M}_2$ in parallel—keeping track of the state of both machines. It will accept a string, then, iff both machines reach a final state on that string.

Let $\mathcal{M}' = \langle Q \times P, \Sigma, \delta', \langle q_0, p_0 \rangle, F_1 \times F_2 \rangle$, where

$$\delta'(\langle q, p \rangle, \sigma) = \langle \delta_1(q, \sigma), \delta_2(p, \sigma) \rangle .$$

Then $\hat{\delta}'(\langle q, p \rangle, w) = \left\langle \hat{\delta}_1(q, w), \hat{\delta}_2(p, w) \right\rangle$. (You should prove this; it is an easy induction on the structure of $w$.) It follows then that

$$\begin{aligned} w \in L(\mathcal{M}') &\Leftrightarrow \hat{\delta}'(\langle q_0, p_0 \rangle, w) \in F_1 \times F_2 \\ &\Leftrightarrow \hat{\delta}_1(q_0, w) \in F_1 \text{ and } \hat{\delta}_2(p_0, w) \in F_2 \\ &\Leftrightarrow w \in L_1 \text{ and } w \in L_2 \\ &\Leftrightarrow w \in L_1 \cap L_2. \end{aligned}$$

$\dashv$

**Corollary 1** *The class or regular languages is closed under relative complement.*

Since $w \in L_1 \setminus L_2$ iff $w \in L_1$ and $w \notin L_2$ iff $\hat{\delta}_1(q_0, w) \in F_1$ and $\hat{\delta}_2(p_0, w) \notin F_2$ iff $\hat{\delta}'(\langle q_0, p_0 \rangle, w) \in F_1 \times (P \setminus F_2)$, we can use essentially the same construction changing only $F'$ to $F_1 \times (P \setminus F_2)$.

**Theorem 5** *The class of regular languages is closed under complement.*

**Proof** (sketch)**:** Following the insight of the corollary, $w \in \overline{L_1}$ iff $w \notin L_1$ iff $\hat{\delta}_1(q_0, w) \notin F_1$. Thus we can let $\mathcal{M}' = \langle Q, \Sigma, \delta_1, q_0, Q \setminus F_1 \rangle$, that is $\mathcal{M}$ with the set of final states complemented wrt $Q$. $\dashv$

Note that if we had proved closure under complement first we could have gotten closure under intersection using DeMorgan's Theorem.

$$L_1, L_2 \text{ reg.} \Rightarrow \overline{L_1}, \overline{L_2} \text{ reg.} \Rightarrow \overline{L_1} \cup \overline{L_2} \text{ reg.} \Rightarrow \overline{\overline{L_1} \cup \overline{L_2}} \text{ reg.} \Rightarrow L_1 \cap L_2 \text{ reg.}.$$

**Definition 45** *A class of languages is closed under* Boolean Operations *iff it is closed under union, intersection, and relative complement.*

**Corollary 2** *Any class of languages closed under relative complement and either union or intersection is closed under Boolean operations.*

**Corollary 3** *The class of regular languages is closed under Boolean operations.*

## 13.2 Using Closure Properties to Prove Regularity

The fact that regular languages are closed under Boolean operations simplifies the process of establishing regularity of languages; in essence we can augment the regular operations with intersection and complement (as well as any other operations we can show preserve regularity). All one need do to prove a language is regular, then, is to show how to construct it from "obviously" regular languages using any of these operations. (A little care is needed about what constitutes "obvious". The safest thing to do is to take the language back all the way to $\emptyset$, $\{\varepsilon\}$, and the singleton languages of unit strings.)

**Example:** Let $L \subseteq \{a, b\}^*$ such that

- '$aa$' never occurs in any string in $L$,
- if '$ab$' occurs anywhere in a string in $L$ then '$ba$' also occurs somewhere in that string.

To show that $L$ is regular, note first that $L$ is the intersection of two languages: one in which only the first property (no '$aa$') is enforced and one in which only the second ('$ab$' implies '$ba$') is.

$$L = L_1 \cap L_2,$$

where $L_1$ is the set of strings over $\{a, b\}$ in which '$aa$' never occurs and $L_2$ is the set in which '$ba$' occurs whenever '$ab$' does.

$$L_1 = \overline{L_3},$$

where $L_3$ is the set of strings over $\{a, b\}$ in which '$aa$' does occur.

$$L_3 = L((a + b)^* aa(a + b)^*).$$

$L_2$ is the set of strings over $\{a, b\}$ in which either '$ab$' does not occur or '$ba$' does ($P \Rightarrow Q \equiv \neg P \vee Q$).

$$L_2 = L_4 \cup L_5,$$

where $L_4$ is the set of strings over $\{a, b\}$ in which '$ab$' never occurs and $L_5$ is the set in which '$ba$' does.

$$L_4 = \overline{L_6}, \qquad L_6 = L((a + b)^* ab(a + b)^*).$$

and

$$L_5 = L((a + b)^* ba(a + b)^*).$$

Thus

$$L = \overline{L_3} \cap (\overline{L_6} \cup L_5)$$

and each of $L_3$, $L_6$ and $L_5$ are regular. Hence $L$ is regular as well.

43. Using this approach, show that the set of strings over $\{a, b\}$ in which the number of '$a$'s is divisible by three but not divisible by two is regular.

44. Starting with simple automata for your "obviously" regular languages and using the constructions of the proofs of the closure properties, build a DFA for this language.

45. Consider the two languages:

    $L_a$**:** The set of strings over $\{a, b\}$ in which the last symbol is not '$b$'.

    $L_b$**:** The set of strings over $\{a, b\}$ in which the last symbol is not '$a$'.

    Using the approach of the previous problem, construct a DFA accepting the language of strings that satisfy both of these descriptions.

46. What is that language? Explain why it is not empty.

## 13.3 Quotient and Prefix

**Definition 46** *The (right)* quotient *of a language $L_1$ wrt $L_2$ (both over $\Sigma$) is*

$$L_1/L_2 \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid (\exists v \in L_2)[wv \in L_1]\}.$$

So the quotient of $L_1$ wrt $L_2$ is the set of prefixes one is left with when one removes suffixes of strings in $L_1$ that are found in $L_2$.

**Example:** Let $L_1 = L(a^*(bc)^*)$ and $L_2 = L((cb^*)^+)$. Then

$$L_1/L_2 = \{w \in \{a, b, c\}^* \mid (\exists v \in L((cb^*)^+))[wv \in L(a^*(bc)^*)]\}.$$

If $wv \in L_1$ then $wv = a^i(bc)^j$ for some $i, j \in \mathbb{N}$.
If $v \in L_2$ then $v = (cb^{k_1})(cb^{k_2}) \cdots (cb^{k_l})$ for some $k_1, k_2, \ldots, k_l \in \mathbb{N},\ l > 0$.
It follows that

$$wv = a^ib(cb)^mc = a^i(cb)^{m-l-1}(cb^1)^{l-1}(cb^0) \text{ and } w = a^ib(cb)^{m-l-1},\ m{-}l{-}1 \geq 0.$$

Thus

$$L_1/L_2 = L(a^*b(cb)^*).$$

47. Let $L_1 = L(a^*ba^*)$ and $L_2 = L(b^*a)$. What is $L_1/L_2$?

48. Let $L_3 = L(ba^*b)$ and $L_1$ remain the same. What is $L_1/L_3$?

**Theorem 6** *The class of regular languages is closed under quotient with* arbitrary languages.

That is, as long as $L_1$ is regular, $L_2$ can be any language whatsoever and $L_1/L_2$ will be regular. (It is not required that it even be possible to effectively decide if a given word is in $L_2$.)

**Proof** (sketch)**:** Let $L_1 = L(\mathcal{M}_1)$ and $\mathcal{M}_1 = \langle Q, \Sigma, \delta, q_0, F\rangle$.
Let $\mathcal{M}' = \langle Q, \Sigma, \delta, q_0, F'\rangle$ where

$$F' = \{q \in Q \mid (\exists v \in L_2)[\hat{\delta}(q, v) \in F]\}.$$

It is easy to show, then, that $w \in L(\mathcal{M}') \Leftrightarrow w \in L_1/L_2$. ⊣

49. Show it.

Note that if it is not possible to effectively decide if $v \in L_2$ then we will not be able to effectively decide if $q \in F'$. But there are only $2^{\mathbf{card}(Q)}$ subsets of $Q$. One of these is the right one. Thus there is *some* DFA that recognizes $L_1/L_2$, although we may not be able to effectively decide which one. Of course, if $L_2$ is regular we can tell if $w \in L_2$ and the construction is effective.

**Corollary 4** *The class of regular languages is closed under the operation*

$$\text{Suffix}(L) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid (\exists v \in \Sigma^*)[wv \in L]\}.$$

Since $\text{Suffix}(L) = L/\Sigma^*$.

50. Suppose $L$ is any nonempty language. What is $\Sigma^*/L$?

51. What is $L/\emptyset$?

52. What is $L/\{\varepsilon\}$?

53. Suppose $L_1$, $L_2$ and $L_3$ are arbitrary languages. Show that

$$L_1/(L_2 \cup L_3) = (L_1/L_2) \cup (L_1/L_3)$$

and that

$$L_1/(L_2 \cap L_3) = (L_1/L_2) \cap (L_1/L_3).$$

54. Suppose, again, that $L_1$, $L_2$ and $L_3$ are arbitrary languages. What is $L_1/(L_2/L_3)$?

55. What is $(L_1/L_2)/L_3$?

## 13.4 Substitution and Homomorphism

Let $f$ be a function that maps each $\sigma \in \Sigma$ to some regular language $L_\sigma$. In general, each $L_\sigma$ may be over its own alphabet $\Gamma_\sigma$ distinct from the others, but we can understand all of the $L_\sigma$ to be languages over $\Gamma = \bigcup_{\sigma\in\Sigma}[\Gamma_\sigma]$. So while the function may map symbols in $\Sigma$ to languages over some other alphabet $\Gamma$, we can take the range of $f$ to be languages over that single alphabet:

$$f : \Sigma \to \boldsymbol{\mathcal{P}}(\Gamma).$$

$f$ is *a substitution of regular languages for* $\Sigma$.

**Definition 47** *If* $w \in \Sigma^*$ *then*

$$f(w) \stackrel{\text{def}}{=} \begin{cases} \{\varepsilon\} & \text{if } w = \varepsilon, \\ f(w') \cdot f(\sigma) & \text{if } w = w' \cdot \sigma. \end{cases}$$

**Definition 48** *If* $L \subseteq \Sigma^*$ *then*

$$f(L) \stackrel{\text{def}}{=} \{f(w) \mid w \in L\}.$$

Note that if $f(\sigma)$ includes $\varepsilon$ then $f$ may erase '$\sigma$'s occurring in strings in $L$, while if $f(\sigma) = \emptyset$ then $f$ has the effect of deleting every string in $L$ in which '$\sigma$' occurs.

**Example:** Let $f(a) = L(a^+ca^+)$ and $f(b) = (b^+cb^+)$. Let $L = ((ab + ba)^*)$. Then $abab \in L$ and

$$\underbrace{acaaa}_{\in f(a)}\underbrace{bbbcb}_{\in f(b)}\underbrace{aaacaa}_{\in f(a)}\underbrace{bcbbb}_{\in f(b)} \in f(w).$$

And

$$\begin{aligned} f(L) &= f(L((ab + ba)^*)) \\ &= (f(\{ab\} \cup \{ba\}))^* \\ &= (f(\{ab\}) \cup f(\{ba\}))^* \\ &= (f(a) \cdot f(b) \cup f(b) \cdot f(a))^* \\ &= (L(a^+ca^+)L(b^+cb^+) \cup L(b^+cb^+)L(a^+ca^+))^* \\ &= L((a^+ca^+b^+cb^+ + b^+cb^+a^+ca^+)^*). \end{aligned}$$

**Theorem 7** *The class of regular languages is closed under substitution by regular languages.*

**Proof** (sketch)**:** If $L$ is regular then $L = L(r)$ for some regular expression $r$. Then $f(L) = L(f(r))$ where $f(r)$ is, in essence, the result of applying $f$ to $r$. $\dashv$

56. Let $L = L(((a + ba)^*ba)^*)$ and $f = \{a \mapsto L(ab^*a), b \mapsto L(b^*ab^*)\}$. Give a regular expression for the language $f(L)$.

**Definition 49** *Let $h : \Sigma \to \Gamma^*$ map symbols of $\Sigma$ to strings over some alphabet $\Gamma$. We say that $h$ is a* homomorphism *of $\Sigma$ to $\Gamma^*$, the* homomorphic image *of $w \in \Sigma^*$ (under $h$) is*

$$h(w) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ h(w') \cdot h(\sigma) & \text{if } w = w' \cdot \sigma \end{cases}$$

*and the* homomorphic image *of $L \subseteq \Sigma^*$ (under $h$) is*

$$h(L) \stackrel{\text{def}}{=} \{h(w) \mid w \in L\}.$$

**Corollary 5** *The class of regular languages is closed under homomorphisms.*

This is because a homomorphism is, in essence, a substitution in which $\mathbf{card}(f(\sigma)) = 1$ for all $\sigma \in \Sigma$. Thus closure under substitution implies closure under homomorphism.

## 13.5 Reversal

**Theorem 8** *The class of regular languages is closed under reversal.*

**Proof** (sketch)**:** Let $L = L(r)$ for some regular expression $r$. Let

$$r^{\mathrm{R}} \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } r = \emptyset \\ \varepsilon & \text{if } r = \varepsilon \\ \sigma & \text{if } r = \sigma \\ (s^{\mathrm{R}} + t^{\mathrm{R}}) & \text{if } r = (s + t) \\ (t^{\mathrm{R}} \cdot s^{\mathrm{R}}) & \text{if } r = (s \cdot t) \\ (s^{\mathrm{R}^*}) & \text{if } r = (s^*). \end{cases}$$

It follows by an easy induction on the structure of $w$ that $w \in L(r) \Leftrightarrow w^{\mathrm{R}} \in L(r^{\mathrm{R}})$. Thus

$$L^{\mathrm{R}} = L(r)^{\mathrm{R}} = L(r^{\mathrm{R}}).$$

$\dashv$

**Proof** (sketch, alternate)**:** Let $L = L(\mathcal{A})$ for some DFA $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$. Let $\mathcal{A}' = \langle Q', \Sigma, \delta', q_0', F'\rangle$ be an *NFA* with:

$$\begin{aligned} Q' &= Q \cup \{q_0'\} \\ \delta'(q_0', \varepsilon) &= F \\ \delta'(q, \sigma) &= \{p \mid \delta(p, \sigma) = q\} \\ F' &= \{q_0\}. \end{aligned}$$

So $\mathcal{A}'$ is, in essence, the NFA one gets by reversing the edges of $\mathcal{A}$.

57. Why is $\mathcal{A}'$ an NFA rather than another DFA.

58. Complete the proof by showing that

$$w \in L(\mathcal{A}) \Leftrightarrow w^{\mathrm{R}} \in L(\mathcal{A}').$$

[**Hint:** Prove first that $\hat{\delta}(q, w) = p \Leftrightarrow q \in \hat{\delta}'(p, w^{\mathrm{R}})$.]

⊣

59. Prove that the class of regular languages is closed under Suffix.
[**Hint:** Why is this question here rather than in Section 13.3?]

## 13.6 Using Closure Results to Prove Languages are Non-regular

Let $L_{ab} = \{a^i b^i \mid i \geq 0\}$. We will take $L_{ab}$ to be our *canonical* non-regular language. We can then use the known closure results for the class of regular languages to prove, by contradiction, that some language $L$ is not regular by showing how to reduce $L$ to $L_{ab}$ using operations that preserve regularity.

**Example:** Let $L = \{w \in \{a, b, c, d\}^* \mid |w|_{ab} \geq |w|_{cd}\}$. To show that $L$ is not regular.
Let $L_1 = L \cap L((ab)^*(cd)^*)$. Then

$$L_1 = \{(ab)^i (cd)^j \mid 0 \leq j \leq i\}.$$

Let $L_2 = h_1(L_1)$ where $h_1 = \{a \mapsto a, b \mapsto \varepsilon, c \mapsto b, d \mapsto \varepsilon\}$. Then

$$L_2 = \{a^i b^j \mid 0 \leq j \leq i\}.$$

Let $L_3 = h_2(L_1)$ where $h_2 = \{a \mapsto b, b \mapsto \varepsilon, c \mapsto a, d \mapsto \varepsilon\}$. Then

$$L_2 = \{b^k a^l \mid 0 \leq l \leq k\}.$$

Then

$$L_2 \cap {L_3}^{\mathrm{R}} = \{a^i b^j \mid 0 \leq j \leq i\} \cap \{a^l b^k \mid 0 \leq l \leq k\} = \{a^i b^j \mid 0 \leq j = i\} = L_{ab}.$$

As the class of regular languages is closed under intersection, homomorphism and reversal if $L$ were regular $L_{ab}$ would be regular as well. But $L_{ab}$ is our canonical *non-regular* language and, consequently, $L$ cannot be regular.

60. Consider again a system of two processes (A and B) exchanging messages as in Exercise 25. Again A sends either '$m_1$' or '$m_2$' and B acknowledges with '$a_1$', '$a_2$' or '$a_{12}$', where '$a_1$' acknowledges '$m_1$', '$a_2$' acknowledges '$m_2$' and '$a_{12}$' acknowledges both. In contrast to Exercise 25, we will now allow any number of '$m_1$'s or '$m_2$'s to be outstanding. We require only that every message is eventually acknowledged and that no acknowledgment is sent unless there is some outstanding message(s) of the corresponding type. Show that the set of finite sequences of messages that satisfy this protocol is *not* regular.
[**Hint:** Start by taking an intersection with a regular set to simplify the language. (Get rid of all the '$m_2$'s, '$a_2$'s, and '$a_{12}$'s.)]

# 14 Some Decision Problems for the Class of Regular Languages

We'll close this study of the regular languages by considering whether certain questions concerning given regular languages can be decided algorithmically. We will focus on a few questions, with you doing a couple more as exercises.

**Membership:** Given a string and a finite representation of regular language, is the string in the language?

**Emptiness:** Given a finite representation of a regular language, is that language empty?

**Finiteness:** Given a finite representation of a regular language, is that language finite?

**Equivalence:** Given finite representations of two regular languages, do they represent the same language?

We will generally assume that the representation used in the instances of these problems are DFAs. This is usually the easiest form to handle and, as we have already established that there are algorithms for translating other representations into the form of DFAs, if the problem can be decided given DFAs it can be decided given any other representation.

61. Why don't the instances of these problems just include the languages themselves rather than representations of the languages?

62. Which of these properties are algorithmically decidable for the class of finite languages?

These problems are all of the type we called "checking problems" in Section I. They are more properly known as *decision problems*: given some instance decide if it satisfies some property. If such a problem can be solved algorithmically the corresponding property is said to be *decidable*.

## 14.1 Membership

*Given a string and a DFA, is the string in the language accepted by the DFA?*

The question here is whether the computation of the DFA on the string terminates in an accepting state. The obvious way of approaching this is to simply simulate the DFA: start with the initial ID, calculate its successor (using the transition function), repeat until a terminal ID is reached, and answer yes iff the terminal ID includes an accepting state. This is an effective procedure—each step can actually be carried out—and it will certainly give the right answer when it finishes. The issue we need to address is whether it will always finish—is it an algorithm? Here we can appeal to our initial discussion of computations in Section 8. If an ID has a successor the length of the remaining input in that successor is exactly one less than the length of the remaining input in the ID. Thus, there are exactly $|w|$ successors in the computation of any DFA on $w$.

63. Which is to say, the length of the computation in transitions (steps) of the DFA is $|w|$. What is the length of the computation in terms of its representation as a sequence of IDs; how many IDs are in the sequence?

64. Can we establish such a bound on the computations of NFAs without $\varepsilon$-transitions? With them?

Consequently, in simulating the DFA, the process of computing the successor will be repeated exactly $|w|$ times. Since strings have finite length, we are guaranteed to reach a terminal ID in a finite number of steps.

**Theorem 9** *Membership is decidable for the class of regular languages.*

It is useful to consider this approach from the perspective of transition graphs as well. In exploring the computation of the DFA on $w$ we are simply following the path labeled $w$ in the transition graph of the DFA that starts at $q_0$. For DFAs there is only one such path and it consists of exactly $|w|$ edges.

## 14.2 Emptiness

*Given a DFA, is the language accepted by that DFA empty?* Membership asks us to decide whether there is an accepting computation on a given input. Emptiness asks us to decide whether there is a accepting computation on *any* input. Since there are infinitely many strings that might be accepted, this is, in general, more difficult: there are systems of computation for which membership (or its equivalent) is decidable but emptiness is not. While we might approach this by applying our algorithm for membership systematically

to all strings over the alphabet of the DFA—starting with the empty string, say, and then all strings of length one, then two, etc.—we cannot check all such strings in finitely much time. For this approach to work we need to identify a finite subset of the strings that suffices: a set we *can* check exhaustively which is guaranteed to include some string in the language if the language includes any string.

We can identify such a subset by thinking back to the Pumping Lemma (Section 11, Lemma 12). This says that there is number $n$ that depends only on the DFA, such that if some string $x$ with length $n$ or more is in the language accepted by the DFA then there is some string shorter than $x$ in the language (the string in which $v$ is pumped zero times). Moreover, the length of that string is no less than $|x| - n$ (since, $|uv| \leq n$, and *a fortiori* $|uv| \leq n$).

Suppose, then, that there is some string of length $n$ or more in the language. Let $w_0$ be such a string with minimal length, i.e. $w_0$ is in the language, $|w_0| \geq n$ and every string with length $n$ or more that is in the language is at least as long as $w_0$. How long is $w_0$ with $v$ pumped zero times? Since this is strictly shorter than $w_0$ and is in the language, and, by choice of $w_0$, every string in the language shorter than $w_0$ is shorter than $n$, it must be the case that $w_0$ with $v$ pumped zero times is shorter than $n$. Thus, we can limit our search to strings of length strictly less than $n$.

All that remains is to figure out what $n$ is for the given DFA. In proving the pumping lemma we used a pigeon hole principle argument to show that the computation of a DFA on any string longer than the number of states (that is $\mathbf{card}(Q)$) must include a loop. Cutting out this loop is what gave us the accepting computation of the DFA on the string with $v$ pumped zero times. Thus, $n$ can be taken to be equal to $\mathbf{card}(Q)$.

The algorithm, then, consists of applying the membership algorithm to all strings over the alphabet of the DFA that are no longer than the size of its state set. If any of these strings are in the language they witness the fact that it is non-empty. If, on the other hand, none of them are, we know as a consequence of the pumping lemma that no longer string is in the language either.

This is even simpler if we think in terms of the transition graph. In that context the emptiness problem is simply asking if there is any path in the graph from the start state to an accepting state. Algorithms for solving this problem (on finite graphs) should be well-known to you (e.g., Dijkstra's or Floyd's algorithms). One of the attractions of representing DFAs as transition graphs is the fact that known graph algorithms can be employed to solve their

decision problems.

## 14.3 Finiteness

*Given a DFA, is the language accepted by that DFA finite?* Just as the emptiness problem can be seen as a (potentially more difficult) generalization of the membership problem, the finiteness problem is, in a particular sense, a generalization of the emptiness problem. Here we need to determine not only if *any* string is in the language accepted by the DFA but *how many* of them there are, in particular, if there are only finitely many of them.

Thinking, again, in terms of the pumping lemma, if there are any strings in the language of length $n$ or greater then there will be infinitely many of them (since we can pump $v$ any number of times). Conversely, if there is no string in the language of length greater than $n$ then there are but finitely many strings in the language (since the number of strings over a given alphabet of length less than $n$ is finite). So again, we can use the membership algorithm, now searching for strings of length $n$ or greater. And again, our problem is to establish an upper bound on the length of the strings we test.

Consider, again, $w_0$, a string of minimal length among those of length $n$ or greater in the language. How long is $w_0$? We have established that, by choice of $w_0$, the length of $w_0$ with $v$ pumped zero times is strictly less than $n$, and, by the hypothesis of the pumping lemma, the length of $v$ is no greater than $n$, we can simply calculate that $n \leq |w_0| < 2n$. Thus we need only search for some string in the language of length between $n$ and $2n$.

65. Give an algorithm for deciding finiteness that is based on known algorithms for deciding problems for graphs.

## 14.4 Equivalence

*Given two DFAs, do they accept the same language?* Here, again, we have a sort of generalization of the emptiness problem. We need not only to establish whether there is any string in the language accepted by a DFA, but whether the set of such strings for one DFA is the same as those accepted by another. In this case the pumping lemma is not much help. Instead, we will appeal to the Myhill-Nerode Theorem (Section 12) and, in particular, the result of Lemma 17 (Section 12.3). This tells us that the result of minimizing a DFA using the construction of Section 12.3 is unique up to isomorphism,

that is to say, is identical to all other minimal size DFAs accepting the same language except, possibly, for the actual names of the states. Isomorphism of edge-labeled graphs isomorphism is another problem for which an algorithm is known (although perhaps not as familiar). We can solve equivalence of DFAs, then, by minimizing them and using the graph algorithm to test isomorphism.

66. Sketch an algorithm to decide isomorphism of DFAs.

We can establish decidability of emptiness even more easily if we combine the closure results of the previous section with earlier results of this section.

67. Suppose $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$. What is $L(\mathcal{A}_1) \setminus L(\mathcal{A}_2)$?

68. Suppose $L(\mathcal{A}_2) \subseteq L(\mathcal{A}_1)$. What is $L(\mathcal{A}_2) \setminus L(\mathcal{A}_1)$?

69. Show that there is an effective construction that, given DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, builds a DFA accepting $L(\mathcal{A}_1) \setminus L(\mathcal{A}_2) \cup L(\mathcal{A}_2) \setminus L(\mathcal{A}_1)$. (This is known as the *symmetric difference* of the languages.) You do not need to give the actual construction, simply show how the constructions of Section 13 can be combined to make such a construction.

70. Use this result, along with decidability of emptiness, to show that equivalence of DFAs is decidable.

We close with a couple of exercises.

71. In Section 13 we established that the *class* of regular languages was closed under reversal: $L$ regular implies $L^{\mathrm{R}}$ regular. Let us say that a *language* $L$ is closed under reversal iff $w \in L$ implies $w^{\mathrm{R}} \in L$. Prove that the question of whether a given regular language is closed under reversal is decidable.
[**Hint:** Use the closure properties and decision procedures we have already established.]

72. Show that decidability of both emptiness and membership is a consequence of decidability of equivalence, i.e., show how an algorithm for equivalence can be used (as a subroutine) to build an algorithm for emptiness or an algorithm for membership.
[**Hint:** For emptiness start out by giving a DFA that accepts the empty language. For membership start out by sketching and algorithm that, given $w$, constructs a DFA accepting $\{w\}$.]