PART II

Finite and Regular Languages

Part II
Finite and Regular Languages

5 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 Σ:*

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

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 Σ.

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 $\emptyset$, ε, and all the singleton languages consisting of just a unit string from Σ 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 Σ 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 Σ is constructible, then we will use that result to establish that every finite language over Σ 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 Σ is constructible as in Definition 18.*

Proof: Suppose L is a singleton language over Σ. 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 $\{v\} \cdot \{\sigma\}$. $\dashv$

Lemma 3 *Every finite language over Σ 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. $\dashv$

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 Σ is infinite—does it still define the class of finite languages? [**Hint:** Surely the fact that Σ is infinite does not make the class of languages that can be constructed any smaller: all finite languages over Σ 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 Σ?]

6 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 Σ:*

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

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

Definition 21 (Regular Expressions) *For any alphabet Σ:*

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

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

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 Σ:*

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

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 'ε' is redundant; every set that can be defined using it can also be defined without it.
[**Hint:** Find a regular expression in which 'ε' 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.

6.1 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*

$$R = S \overset{\text{def}}{\iff} L(R) = L(S).$$

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)	$R\varepsilon = \varepsilon R = R$	*(ε 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 6 and Example 6 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,

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

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

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

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.]

6.2 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 $w = w_1w_2$ 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 $w = w_1aaw_2$ 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, $w_1aa \in L_{\overline{bb}}$. ⊣

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 ε, 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

$$x_1x_2 \in L((baa^*)^n) \cdot L(baa^*) = L((baa^*)^{n+1})$$

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

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

or

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

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

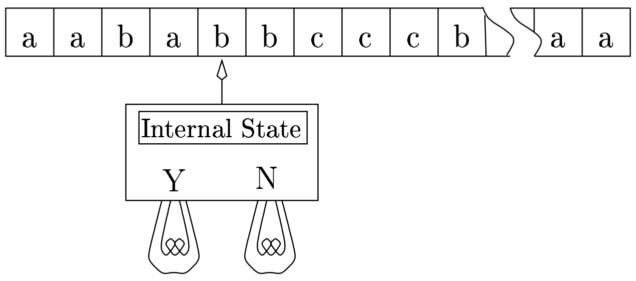
Finally, we put these together and get

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

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.

7 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*,*
Σ	*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 δ, it will generally suffice to specify only δ, the start state and the set of final states.

7.1 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)

$$\langle q, w \rangle \underset{\mathcal{A}}{\vdash} \langle p, v \rangle \overset{\text{def}}{\Longleftrightarrow} w = \sigma v \textit{ and } p = \delta(q, \sigma).$$

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 δ is a total function, every ID in which $w \neq \varepsilon$ has a successor and that successor is unique. (Thus, $\underset{\mathcal{A}}{\vdash}$ 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 $i > 0$, $\langle q_{i-1}, w_{i-1} \rangle \underset{\mathcal{A}}{\vdash} \langle q_i, w_i \rangle$ and which is closed under $\underset{\mathcal{A}}{\vdash}$: for all i, if $w_i \neq \varepsilon$ and $\langle q_i, w_i \rangle \underset{\mathcal{A}}{\vdash} \langle q_{i+1}, w_{i+1} \rangle$ then $\langle q_{i+1}, w_{i+1} \rangle$ is included in the sequence.*

Since $\underset{\mathcal{A}}{\vdash}$ 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, $w_{|w_1|+1}$ will be ε. Thus, all computations of a DFA halt and they all take exactly $|w_1|$ steps.

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

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

$\langle q, w\rangle \underset{\mathcal{A}}{\overset{n}{\vdash}} \langle p, v\rangle$ ("$\langle q, w\rangle$ 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^{th} *successor of* $\langle q, w\rangle$*.)*

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

Lemma 5 $\underset{\mathcal{A}}{\overset{*}{\vdash}}$ *is the reflexive, transitive closure of* $\underset{\mathcal{A}}{\vdash}$.

Proof: ($\overset{*}{\vdash}$ extends $\vdash$:)

$\langle q, w\rangle \underset{\mathcal{A}}{\overset{1}{\vdash}} \langle p, v\rangle$ iff $\langle q, w\rangle \underset{\mathcal{A}}{\vdash} \langle p, u\rangle$. (By definition of a computation.)

($\overset{*}{\vdash}$ is reflexive:)

$\langle q, w\rangle \underset{\mathcal{A}}{\overset{0}{\vdash}} \langle q, w\rangle$.

($\overset{*}{\vdash}$ is transitive:)

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

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*

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

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

The following is another intuitively obvious lemma, but one that is frequently useful and deserves to be established rigorously.

Lemma 6 *For all $w, v \in \Sigma^*$ and $q, p \in Q$,*

$$\langle q, w\rangle \vdash^* \langle p, \varepsilon\rangle \Leftrightarrow \langle q, wv\rangle \vdash^* \langle p, v\rangle .$$

This just points out that the initial portion of a computation is completely independent of the unscanned part of the input: all strings that share a prefix compute exactly the same sequence of IDs on that prefix.

Proof: We will prove both the 'if' and the 'only if' directions at the same time (violating our general rule—the reason will be evident below). Proceeding by induction on the length of the computation:

(Basis:)

$$\langle q, w\rangle \vdash^0 \langle p, \varepsilon\rangle \Leftrightarrow \quad q = p \text{ and } w = \varepsilon \quad \Leftrightarrow \langle q, wv\rangle \vdash^0 \langle p, v\rangle .$$

(IH:)

Suppose the lemma is true of all computations of length n or less.

(Ind:)

Suppose $\langle q, w\rangle \vdash^{n+1} \langle p, \varepsilon\rangle$. Then there is some $w_1 \in \Sigma^*$, $\sigma \in \Sigma$ and $p' \in Q$ such that $w = w_1\sigma$ and

$$\langle q, w_1\sigma\rangle \vdash^n \langle p', \sigma\rangle \vdash \langle p, \varepsilon\rangle .$$

By the '$\Leftarrow$' direction of the IH, $\langle q, w_1\rangle \vdash^n \langle p', \varepsilon\rangle$.

By the '$\Rightarrow$' direction of the IH, $\langle q, w_1\sigma v\rangle \vdash^n \langle p', \sigma v\rangle \vdash \langle p, v\rangle$.

Which is, $\langle q, wv\rangle \vdash^{n+1} \langle p, v\rangle$.

For the other direction, suppose $\langle q, wv\rangle \vdash^{n+1} \langle p, v\rangle$. Then, again, there is some $w_1 \in \Sigma^*$, $\sigma \in \Sigma$ and $p' \in Q$ such that $w = w_1\sigma$ and

$$\langle q, w_1\sigma v\rangle \vdash^{n} \langle p', \sigma v\rangle \vdash \langle p, v\rangle\,,$$

and, by the definition of $\vdash$, it must be the case that $\delta(p', \sigma) = p$. Proceeding, again, with, first, the '$\Leftarrow$' direction of the IH and then the '$\Rightarrow$' direction, $\langle q, w_1\rangle \vdash^{n} \langle p', \varepsilon\rangle$ and then $\langle q, w_1\sigma\rangle \vdash^{n} \langle p', \sigma\rangle$. Moreover, $\langle p', \sigma\rangle \vdash \langle p, \varepsilon\rangle$, since $\delta(p', \sigma) = p$. Thus,

$$\langle q, w_1\sigma\rangle \vdash^{n} \langle p', \sigma\rangle \vdash \langle p, \varepsilon\rangle\,.$$

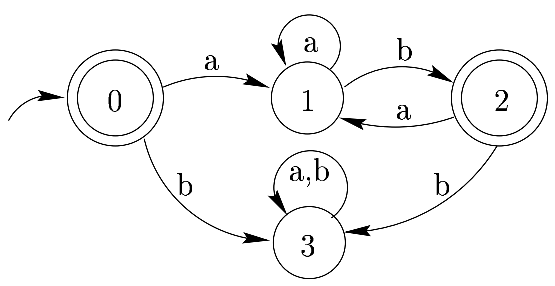
$\dashv$

(The idea, here, is that we use one direction of the IH to break the computation and the other to splice it back together. Note that, even though we prove both directions with a single induction, we prove them separately them in the inductive step.)

7.2 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:

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



[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 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 δ be total corresponds to a requirement that every node in the graph has an out-edge for each symbol in Σ. It follows that every string over Σ 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 δ 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 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 δ, the "edge" function.

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

- $\hat{\delta}(q, \varepsilon) = q$, *for all* $q \in Q$.
- *If* $q \in Q$, $w \in \Sigma^*$ *and* $\sigma \in \Sigma$ *then* $\hat{\delta}(q, w\sigma) = \delta(\hat{\delta}(q, w), \sigma)$.
- *Nothing Else.*

This just says that from any node the path of zero length (labeled ε) 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 'σ'.

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

Just as the directly computes relation captures, in essence, the meaning of δ, 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 δ. 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 δ. Why, then, don't we define DFAs in terms of $\hat{\delta}$ instead of δ?

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 Σ^*,

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

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

7.3 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). While 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, '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 '$\mathrm{A}_1\mathrm{B}_1$'. If q is '$\mathrm{A}_2\mathrm{B}_1$', on the other hand, we will complete the pending 'A'—we go to state 'B_1'. And if q is '$\mathrm{A}_1\mathrm{A}_1$' 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 'ε'. 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 'ε'; the set of final states is just $\{\varepsilon\}$.

So our choice of state sets yields the automaton of Figure 1(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 'ε' that ends in a final state (i.e., in 'ε') 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 'ε' back to 'ε'. 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.

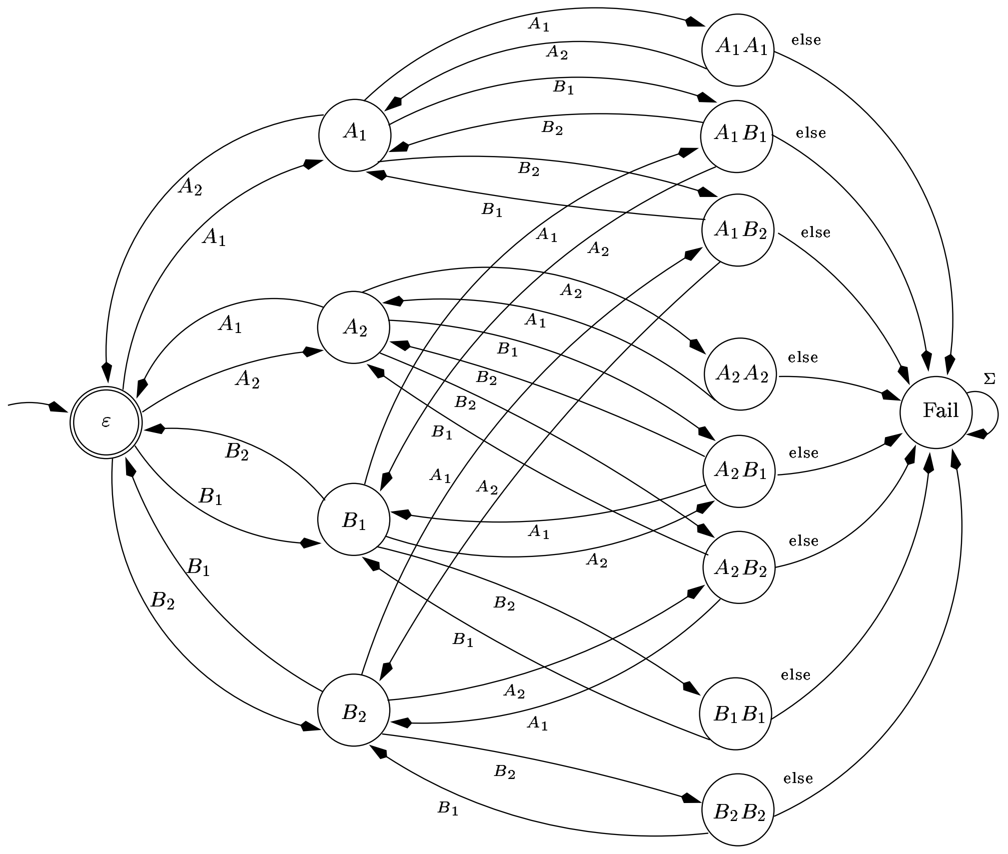


Figure 1: $\mathcal{A}_2$: Automaton for feasible schedules.

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

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

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

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

$$\begin{array}{l} |w|_{A_1} - |w|_{A_2} = |q|_{A_1} - |q|_{A_2} \\ |w|_{B_1} - |w|_{B_2} = |q|_{B_1} - |q|_{B_2} \end{array}$$

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 $\hat{\delta}(\varepsilon, w) = \varepsilon \neq \text{Fail}$ 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 2.)

The way the table should be read is:

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

$\hat{\delta}(\varepsilon, w)$	$\hat{\delta}(\varepsilon, w')$	σ	$\lvert w'\rvert_{A_1} - \lvert w'\rvert_{A_2}$	$\lvert w'\rvert_{B_1} - \lvert w'\rvert_{B_2}$	w' Ovr	$\lvert w\rvert_{A_1} - \lvert w\rvert_{A_2}$	$\lvert w\rvert_{B_1} - \lvert w\rvert_{B_2}$	w Ovr
ε	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	ε	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	ε	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	ε	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	ε	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	✓
	$Fail$	Σ	—	—	✓	—	—	✓

Figure 2: Proof of Lemma 7.

> *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 σ, 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 δ, 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 is called proof by *simultaneous induction.* Note also that even if all we wanted to establish was the invariant for 'ε', 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. $\dashv$

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. It is *not* necessary to use it. Note, though, the inductive step has a case for each state in Q and for each edge incident on that state. This can make for a long proof. As each row of the table carries out the proof for one case, it serves to present the proof in a concise and organized way. 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 δ 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.,

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