Notes on Finite State Machines and Regular Languages

J. Todd Wilson

Andrew Clifton

Department of Computer Science

California State University, Fresno

Fresno, CA 93740

June 4, 2015

twilson@csufresno.edu, andyclifton@mail.fresnostate.edu

As the major concerns of computer science are *computation* and the nature and variety of the *machines* that support it, in all their glorious but unwieldy variety, a particularly productive strategy in the study of computer science is to focus on special cases of computations (and machines) that exhibit two opposing qualities: they are special enough to make a detailed study feasible, yet they are general enough to provide useful insight into many broader questions. In the whole of computer science, there is no better example of a subject that meets these two requirements than the one involving Finite State Machines and Regular Languages.

These notes present the standard definitions, techniques, and results of this subject, although in occasionally new ways. They are currently in draft form and contain parts in various stages of completion, from bare outlines to polished text, so please do not distribute them without permission of the author.

Introduction

Suppose we are given a string of characters, say `twilson@csufresno.edu`, and need to determine whether this string represents a valid email address. Or we are given a string and need to determine whether it represents a valid date, or zip code, or system log entry, or programming-language statement. These are all examples of *string recognition problems*, and these problems form the context of our study.

What do we mean by "valid"? Clearly we must have some definition in mind of what strings are, and hence, are not, part of the class we are interested in. At the most basic level, we need to know what *characters* or, more generally, what *symbols* are allowed in our strings. We refer to the set of all valid symbols as the *alphabet* and write it Σ. For example, for email addresses Σ includes not just alphanumeric characters, but also `.` `@` and any other characters allowed in email addresses.

See RFC ... for all the gruesome details on what characters are permissible in email addresses.

Sometimes we need to refer to the set of *all* strings, of any length ≥ 0 over an alphabet. We write this as Σ^*.

A *language* is some subset of Σ^* containing only the strings we are interested in. For example, the language L_{email} contains only the valid email addresses. Formally speaking, a language is a possibly-infinite set of strings, but as we

will see, you can also think of a language as a mechanism for either *recognizing* strings in that set, or for *generating* all strings in that set.

Again, suppose we are interested in the language L_{email}. How can we describe what constitutes a valid email address? By looking at the above example `twilson@csufresno.edu`, we can see that an email consists some *user identifier*, follow by an `@`-sign, follow by a hostname. We can formalize this by giving it as a *grammar* G_{email}:

$$\begin{aligned} \langle \textit{email} \rangle &\to \langle \textit{user identifier} \rangle @ \langle \textit{hostname} \rangle \\ \langle \textit{user identifier} \rangle &\to \ldots \\ \langle \textit{hostname} \rangle &\to \ldots \end{aligned}$$

(Obviously, we still need to fill in the definitions of $\langle \textit{user identifier} \rangle$ and $\langle \textit{hostname} \rangle$.)

Formally speaking, a grammar is a four-tuple (V, Σ, P, S) where V is a finite set of *non-terminals* (e.g., $\langle \textit{email} \rangle$, $\langle \textit{hostname} \rangle$), Σ is the alphabet, the set of terminals, $P \subset V \times (V \cup \Sigma)^*$ is a finite set of *rules*, and $S \in V$ is the *start symbol*.

This definition actually describes *context-free* grammars, those in which the left-hand-side of every rule is restricted to a single nonterminal. There are grammars, which we will consider later, in which this restriction does not hold; e.g., $\texttt{a}\langle B \rangle \texttt{c} \to \texttt{abc}$ is a valid, but not context-free, rule.

Informally, a grammar is defined by its rules, and the nonterminals and terminals that occur in it (the above definition allows for grammars to have nonterminals and terminals that are never used by the rules, but since these have no effect, we can safely ignore them). Every rule describes the *structure* of a particular nonterminal, as a sequence of nonterminals and/or terminals. Note that it *is* valid for this sequence to be empty! A rule $\langle A \rangle \to \varepsilon$ indicates that the definition of $\langle A \rangle$ is empty, and is refered to as an *epsilon rule*. Also note that there is no restriction requiring each nonterminal to have a single defining rule. For example, this is a perfectly valid grammar:

$$\begin{aligned} \langle A \rangle &\to \texttt{a} \\ \langle A \rangle &\to \texttt{b} \end{aligned}$$

This expresses the fact that the nonterminal $\langle A \rangle$ can recognize or match *either* a literal `a` *or* a literal `b`.

When presenting grammars, we will normally assume that the *first* rule is the start rule.

String recognition with a grammar

Here we informally build up a sketch of an algorithm for doing string recognition using a grammar. Consider, again, the example of an email address. We want to use the grammar G_{email} to determine whether `"twilson@csufresno.edu"` is a valid email address.

1. We begin with the start rule $\langle \textit{email} \rangle$. Its definition tells us that a valid email

address consists of a ⟨*user identifier*⟩ followed by a literal `@` followed by a ⟨*hostname*⟩.

2. So we recursively ask whether ⟨*user identifier*⟩ matches some prefix of `"twilson@csufresno.edu"`. Presumably (hopefully!) it matches `twilson`, leaving the remainder of the string as `@csufresno.edu`.

3. We now ask whether the literal `@` matches some prefix of `@csufresno.edu`; it does, with the remainder of the string now being `csufresno.edu`.

4. We recursively ask whether ⟨*hostname*⟩ matches a prefix of `csufresno.edu`. It does, and all that's left is the empty string ε.

5. Since the start rule accepted the string given, and since there's nothing left of the string, the string is accepted as a valid email address.

Note that *both* the final conditions are important: the entire definition of the start rule must have been successful (e.g., if the input string was `twilson|csufresno.edu` the `@` would have failed to match), *and* there must be nothing left of the string when we are finished.

Above we assumed that the definition of a rule could be followed from left to right, but this is not required. It's not hard to construct grammars for which this method will result in an infinite loop:

$$\langle A \rangle \to \langle A \rangle \texttt{a}$$
$$\langle A \rangle \to \varepsilon$$

More on derivations...

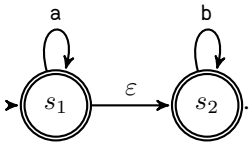
Non-deterministic machines

ε-transitions

We've seen how DFAs can be used to recognize and generate strings, and now we want to consider the question of whether, and if so, how DFAs can be *combined* to form larger machines. We'll see later on that normal DFAs can in fact be combined to form the union, concatenation, etc. of their languages (and, indeed, this result will be a key part of our proof that DFAs recognize accept exactly the regular languages) but performing these operations on standard DFAs is somewhat tricky involved. In this section, we'll introduce an extension to DFAs called NDA-ε, DFAs with a touch of *non-determinism*

"Non-determinism" is a computer science term meaning "magic".

A NDA-ε is a DFA in which transitions can be labeled with an element of $\Sigma \cup \{\varepsilon\}$. That is, we now allow transitions that "recognize" the empty string! For example, here is a NDA-ε that recognizes the language $\texttt{a}^*\texttt{b}^*$:



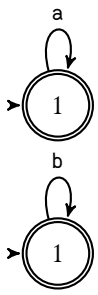
This looks crazy; how does the machine know when to follow the ε arc? In fact, we redefine acceptance for NDA-ε machines as follows

Definition 0.1 An NDA-ε accepts a string if there is *any* execution of the machine in which the machine terminates in a final state with the entire string consumed.

That is, a NDA-ε *magically* knows when to take an ε-transition, so that if there is any possible way of ending in a final state, it will do so. Later, we'll see how this can be implemented on normal, non-magical computers, and indeed how every NDA-ε can be transformed into an equivalent DFA. Right now, whether ε-transitions are *useful*; we'll worry about how to actually make them work later.

Combining DFAs with ε-transitions

Here are the DFAs that accept `a`* and `b`*:



If we look at the above machine, it would appear that all we have to do to construct the concatenation of two machines is put an ε-transition between them. In fact, that's almost the case. For what follows, we require our machines to have a certain structure:

- The machine's start state must have a *single* transition from it, and this must be an ε-transition.
- The machine must have a single final state, and this state must have a single ε-transition into it.

Any NFA can be trivially transformed into one that meets these criteria. If the start state is not suitable then we simply add a new start state, with a single ε-transition to the original start state. Likewise, if the final state(s) are not suitable, we simply add a new final state, and add ε-transitions from every existing final state (which will no longer be final) to the new final state. Thus, every machine is of the form:

Given this assumption, we can now proceed to construct machines for each of the "constructors" of regular sets:

Empty set

Single symbol `a`

Concatenation (of machines M_1 and M_2)

Union (of machines M_1 and M_2)

Kleene Star

Execution of a NDA-ε

We now come to the problem of actually "running" a NDA-ε. When we execute a DFA, for any combination of state s and input symbol a we know that $\delta(s, a)$ will have 0 or 1 values (assuming we are using the partial-function/trap-state simplification). Now, we also have to include any values of $\delta(s, \varepsilon)$, of which there may be 0, 1, or n (n finite). As with our DFA execution function, we will have `delta` return a list of states:

- If the list is empty, we reject the input immediately.
- If the list contains a single state then we transition to it.
- What do we do if the list contains more than 1 elements?

To make things clear, we are dealing with a situation such as

example machine

At this point in the execution, we don't know which of the following states, if any, will lead to a successful final state. *So we try all of them.* We extend $\delta(s, a)$ to $\Delta(S)$ where S is a *set* of states, each of the form (s, w). $\Delta(S)$ is defined as

$$\begin{aligned}\Delta(S) =& \{(s', w) \mid (s, aw) \in S \wedge \delta(s, a) = s'\} \cup \\ & \{(s', w) \mid (s, w) \in S \wedge \delta(s, \varepsilon) = s'\}\end{aligned}$$

That is, it is the set of states that we could get to from any state in S by either accepting another character from the string, or by following an ε-transition.

Non-Deterministic Finite Automata

NDA-ε machines are in fact just an instance of *non-deterministic* finite automata. An NDA is an FDA in which multiple arcs with the same label are allowed out of a state:

example machine

Given the preceding discussion of ε-transitions, it should now be obvious how this works: we redefine acceptance so that a string is accepted if there is *any* sequence of states that leads to a final state, and when executing a NDA we simply try *all* the possible transitions.

Representations of Regular Languages

We've seen several "representations" of languages: regular expressions, regular grammars, and the languages defined by DFAs and NDAs. Here we will consider conversions between all of these representations. In particular, by showing that every NDA can be converted to an equivalent DFA, we will prove that non-determinism does *not* offer any additional computational power, and by showing that there is a bijection between DFAs and REs we will show that DFAs (and hence, NDAs) accept exactly the regular languages. (In fact, we have been dealing with regular languages all along, just in various forms.)

Regular Grammar to DFA

Regular Expression to NDA-ε

We've seen most of this conversion above. Here we will just present the results.

NDA-ε to DFA

NDA to DFA

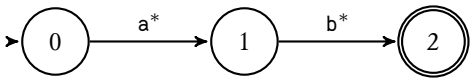
NDA/DFA to Regular Expression (direct)

We will examine two methods for converting an NDA or DFA to a regular expression. The first method is relatively easy to understand and perform by hand, but harder to implement. The second method has a straightforward and efficient implementation, but is harder to understand and *much* harder to perform by hand.

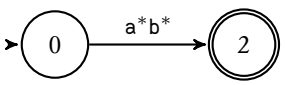
Incidentally, these results, together with the proof that every RE defines a regular language, prove that every NDA/DFA recognizes a regular language.

Method 1: Expression Graphs

An *expression graph* is, like a DFA/NDA, a directed graph with a distinguished initial node and a set of final nodes. Here, however, the arcs between nodes are labeled with regular expressions. For example:

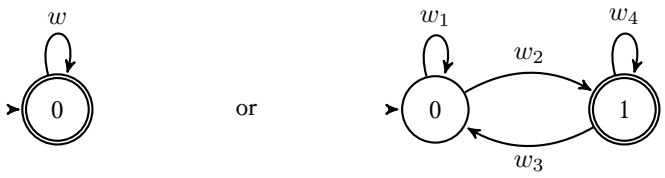


This expression graph accepts the language $\texttt{a}^*\texttt{b}^*$. Since edges can be labeled with arbitrary regular expressions, we can simplify this graph by *deleting* state s_1, replacing it with a single arc from s_0 to s_1, labeled with the concatenation of $\texttt{a}^*$ and $\texttt{b}^*$:



Our general algorithm will thus be to choose an arbitrary non-initial, non-final state to delete, examine its inbound and outbound arcs, and combine them into arcs *directly* connecting the now-adjacent nodes (i.e., bypassing the deleted

node). Once the deleted node has been fully disconnected, it can be removed from the graph. Eventually, we will have a graph in one of two forms



The first represents the regular expression w^* while the second represents $w_1^* w_2 (w_3 + w_4 w_1^* w_2)^*$. (Although this might appear complex, in practice the regular expression identities in figure **??** can often be used to significantly simplify the resulting expression.)

Method 2: Systems of Linear Equations

Regular Expression to DFA (direct)

We mentioned above that it is possible, but cumbersome, to directly construct the union and concatenation of a pair of DFAs, and the Kleene star of a single DFA. Since this construction gives us a *directly* translation from RE to DFA it serves to emphasize that all regular languages can be accepted by a DFA.

Introduction

Suppose we are given a string of characters, say `twilson@csufresno.edu`, and need to determine whether this string represents a valid email address. Or we are given a string and need to determine whether it represents a valid date, or zip code, or system log entry, or programming-language statement. These are all examples of *string recognition problems*, and these problems form the context of our study.

We will be studying simple machines that do nothing but read strings, a character at a time, from left to right, indicating after reading each character whether or not they *accept* (or *recognize*) the string read so far. Each such machine will have a fixed, finite amount of "memory" it can use to remember information about the characters it has read, or "reference" information it needs to use, to help it decide whether or not to accept the string it has read up to that point. It is not allowed to look ahead at the characters it has yet to read, nor look back at or re-read the characters it has already read. And it is completely *deterministic*, in that it always produces the same result when given the same string as input. For reasons that will (eventually) become clear, we call such machines *Deterministic Finite Automata* or DFAs.

If you'd like to have a picture in mind, consider a DFA to be a black box that has access to a *tape*, onto which is written, in distinct cells from left to right, the characters of an *input string*. The machine accesses this tape by means of a *read head*, which is always positioned either just to the left of one of the characters on the tape or at the end. The box also has a *light*, its only means

of output, which is either on or off at any given time. Finally, the box has two buttons: a "reset" button that moves the read head back to the beginning of the tape and puts the machine in its *initial state*, which includes turning its light on or off; and a "read" button that causes the machine's read head to pass over and read the next character to its right (the only way the machine gets information about this character), and adjust the status of its light. The machine *accepts* a particular string when, after we write the string on the tape, push the "reset" button, and then push the "read" button for each character in the string (if any), the machine has its light on at the end of the process.

Note that, as I've described it, it makes sense to consider whether or not our machine accepts the *empty string*, the string with no characters and thus length 0: it does so if its light goes on when the machine is reset. It also makes sense to consider whether our machine accepts the various *prefixes* of the input string, since its light is on or off at the end of reading each prefix, and the rest of the characters have no influence on what the machine has done up to that point.

Although I've described the acceptance process of a DFA in physical terms, its input/output behavior, being deterministic, is completely characterized by the set of strings it accepts. Taking advantage of a linguistic metaphor, I will use the term *language* to mean any sets of strings; thus, every machine accepts a particular language. One important component of our study will be to understand the class of languages accepted by DFAs. We will call these languages *regular*.

Examples of the kinds of questions we would like to answer about DFAs and regular languages are:

- Given a set of strings, is it regular? In other words, is there a DFA that accepts exactly these strings?
- Are there other ways of representing regular languages besides using DFAs that might be more convenient for other purposes?
- What are the closure properties of regular languages? In other words, what operations, when performed on regular languages, always result in regular languages? (I'm thinking of such operations as union, intersection, and difference, as well as many more complicated ones we'll define later.) How can the machines associated with these languages be constructed?
- Are there extensions to the notion of DFA we can make that do not change the class of languages they accept? This would allow us to investigate "higher-level machines" that might be easier to construct or work with tha DFAs but still accept the same languages, the same way that higher-level programming languages make it easier to write programs than assembly languages do, even though they don't allow us to write any new programs.
- How can we show that a particular language is *not* regular? It's one thing to show that a language is regular—you just need to construct a DFA that

accepts it—but it's something altogether different to show that a language is not regular: you have to show somehow that, no matter how clever you are, you will never be able to construct an accepting DFA.

- How much of the theory of DFAs and regular languages can be implemented on a computer (for example in Haskell)?
- What are the applications of DFAs to other areas of computer science? In particular, how can we use DFAs and regular languages to solve string-recognition problems, which come up all over the place?
- What other directions are there that can be explored further?

Basic Definitions

We now begin the formal treatment of our subject. The initial goal is to capture the informal notions introduced in the previous section as precise, mathematical defintions, most of which are inductive, and to prove our results precisely, mostly by induction.

Characters and Strings

We start with a fixed, finite set Σ of *characters*, which we will use to build our strings. We call Σ the *alphabet*.

Definition 0.2 (Strings) We define the set Σ^* of *strings* over Σ by the following rules:

$$\textsc{Empty}\ \frac{}{\varepsilon \in \Sigma^*} \qquad \textsc{Cons}\ \frac{a \in \Sigma \quad w \in \Sigma^*}{aw \in \Sigma^*}$$

Here, ε stands for the empty string, and aw is the result of adding the character a to the beginning of the string w to get a string whose length is one greater.

Definition 0.3 (Length and Concatenation of Strings) The length, $|w|$, of a string w and the concatenation, $w \cdot u$, of two strings w and u are defined by recursion on w:

$$|\varepsilon| = 0, \qquad \text{(LenEmpty)}$$
$$|aw| = 1 + |w|; \qquad \text{(LenCons)}$$
$$\varepsilon \cdot u = u, \qquad \text{(CatEmpty)}$$
$$(aw) \cdot u = a(w \cdot u). \qquad \text{(CatCons)}$$

Theorem 0.4 *(Length of concatenation) For all $w, u \in \Sigma^*$, $|w \cdot u| = |w| + |u|$.*

Proof. By induction on w.

Case EMPTY: $w = \varepsilon$. For all $u \in \Sigma^*$, we have

$$\begin{aligned} |\varepsilon \cdot u| &= |u| && \text{(by CATEMPTY)} \\ &= 0 + |u| \\ &= |\varepsilon| + |u| && \text{(by LENEMPTY)} \end{aligned}$$

Case CONS: $w = aw'$, where $a \in \Sigma$ and $w' \in \Sigma^*$. For all $u \in \Sigma^*$, we have

$$\begin{aligned} |(aw') \cdot u| &= |a(w' \cdot u)| && \text{(by CATCONS)} \\ &= 1 + |w' \cdot u| && \text{(by LENCONS)} \\ &= 1 + |w'| + |u| && \text{(by the IH)} \\ &= |aw'| + |u| && \text{(by LENCONS)} \end{aligned}$$

⊣

Theorem 0.5 *(Identity and associativity of concatenation) For all $w, u, v \in \Sigma^*$,*

1. $w \cdot \varepsilon = w$
2. $w \cdot (u \cdot v) = (w \cdot u) \cdot v$

PROOF. I'll prove these together by induction on w, although they could just as easily be proven individually the same way.

Case EMPTY: $w = \varepsilon$. Then $\varepsilon \cdot \varepsilon = \varepsilon$ by (CATEMPTY), and for every $u, v \in \Sigma^*$, we have $\varepsilon \cdot (u \cdot v) = u \cdot v = (\varepsilon \cdot u) \cdot v$ by two instances of (CATCONS).

Case CONS: $w = aw'$, where $a \in \Sigma$ and $w' \in \Sigma^*$. Then $(aw') \cdot \varepsilon = a(w' \cdot \varepsilon) = aw'$ by (CATCONS) and the IH, and for all $u, v \in \Sigma^*$, we have

$$\begin{aligned} (aw') \cdot (u \cdot v) &= a(w' \cdot (u \cdot v)) && \text{(by CATCONS)} \\ &= a((w' \cdot u) \cdot v) && \text{(by the IH)} \\ &= (a(w' \cdot u)) \cdot v && \text{(by CATCONS)} \\ &= ((aw') \cdot u) \cdot v && \text{(by CATCONS)} \end{aligned}$$

⊣

Languages and Language Operations

As I mentioned in the introduction, I'll use the term *language* to mean any set of strings. Since Σ^* is the set of all strings (over our fixed alphabet Σ), saying that L is a language is the same as saying $L \subseteq \Sigma^*$, which makes Σ^* the maximum, or largest, language. On the other end of the spectrum, the empty set, $\varnothing$, is the minimum, or smallest, language. Next up in size are the languages that contain one string, that is $\{w\}$ where $w \in \Sigma^*$, which includes the language $\{\varepsilon\}$. Note that the size of the strings in a language (their lengths) has nothing to do with the size of the language itself (its cardinality as a set).

Because languages are sets, all of the usual set operations can be performed on languages and result in languages:

- $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$ (union),
- $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$ (intersection),
- $L_1 \setminus L_2 = \{w \mid w \in L_1 \wedge w \notin L_2\}$ (set difference).

More generally, if $\phi(w)$ is any property of strings, then $\{w \mid \phi(w)\}$ is a language, so we can define languages such as $\{w \mid w$ is the password of an NSA executive$\}$ without even knowing what strings are included! However, there are two specific operations on languages that are especially important for our study—called *language concatenation* and *Kleene star* and defined shortly—which arise because the components of our languages, namely strings, themselves have an internal structure involving characters and concatenation.

Definition 0.6 (Concatenation of languages) If $L_1, L_2 \subseteq \Sigma^*$, then we define the *concatenation* of L_1 and L_2 to be the language

$$L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}.$$

That is,

$$w \in L_1 \cdot L_2 \quad \text{if and only if} \quad \exists w_1 \exists w_2 \; w_1 \in L_1 \wedge w_2 \in L_2 \wedge w = w_1 \cdot w_2,$$

or, in other words, $w \in L_1 \cdot L_2$ precisely when we can split w up into two parts (in at least one but possibly more than one way), so that the first part is in L_1 and the second part is in L_2.

The following theorem lists some properties of this operation:

Theorem 0.7 *(Properties of language concatenation)*

1. $\varnothing \cdot L = L \cdot \varnothing = \varnothing$
2. $\{\varepsilon\} \cdot L = L \cdot \{\varepsilon\} = L$
3. $L_1 \cdot (L_2 \cdot L_3) = (L_1 \cdot L_2) \cdot L_3$
4. $L_1 \subseteq L_2$ *implies* $L \cdot L_1 \subseteq L \cdot L_2$ *and* $L_1 \cdot L \subseteq L_2 \cdot L$
5. $L \cdot (L_1 \cup L_2) = (L \cdot L_1) \cup (L \cdot L_2)$ *and* $(L_1 \cup L_2) \cdot L = (L_1 \cdot L) \cup (L_2 \cdot L)$
6. $L \cdot (L_1 \cap L_2) \subseteq (L \cdot L_1) \cap (L \cdot L_2)$ *and* $(L_1 \cap L_2) \cdot L \subseteq (L_1 \cdot L) \cap (L_2 \cdot L)$, *but not necessarily the reverse.*

Proof. I'll prove the first part of (6); the proofs of the rest are left as exercises for the reader.

Let $L, L_1, L_2 \subseteq \Sigma^*$ be arbitrary, and assume $w \in L \cdot (L_1 \cap L_2)$. Then by Definition 0.6 and the comment afterward, $w = w_1 \cdot w_2$ for some $w_1, w_2 \in \Sigma^*$

with $w_1 \in L$ and $w_2 \in L_1 \cap L_2$. By the definition of intersection, this means $w_2 \in L_1$ and $w_2 \in L_2$. Therefore $w_1 \cdot w_2 \in L \cdot L_1$ and $w_1 \cdot w_2 \in L \cdot L_2$, showing that $w = w_1 \cdot w_2 \in (L \cdot L_1) \cap (L \cdot L_2)$.

Why isn't the reverse necessarily true? Intuitively, if $w \in (L \cdot L_1) \cap (L \cdot L_2)$, then $w \in L \cdot L_1$ and $w \in L \cdot L_2$, and so we can write $w = w' \cdot w_1$ with $w' \in L$ and $w_1 \in L_1$, and write $w = w'' \cdot w_2$ with $w'' \in L$ and $w_2 \in L_2$, but there is no guarantee that $w' = w''$ or $w_1 = w_2$, which we would need in order to claim $w \in L \cdot (L_1 \cap L_2)$. However, to really disprove this, we would need to find a *counterexample*, so here is one that realizes this intuitive idea: let $L = \{\mathtt{a}, \mathtt{aa}\}$, $L_1 = \{\mathtt{ab}\}$, and $L_2 = \{\mathtt{b}\}$; then the string $\mathtt{aab}$ is in both $L \cdot L_1$ and $L \cdot L_2$ (as $\mathtt{a} \cdot \mathtt{ab}$ and $\mathtt{aa} \cdot \mathtt{b}$, respectively) but is not in $L \cdot (L_1 \cap L_2)$, since $L_1 \cap L_2 = \varnothing$. ⊣

Note that, according to Theorem 0.7.1–2, $\varnothing$ is a *zero element* for concatenation, and $\{\varepsilon\}$ is a *unit element* for concatenation. That is, in the numerical analogy where concatenation is multiplication, $\varnothing$ acts like 0 and $\{\varepsilon\}$ acts like 1. For this reason, we will sometimes denote these two sets 0 and 1, respectively. However, be careful with this analogy: although concatenation is associative (Theorem 0.7.3), is not commutative ($L_1 \cdot L_2$ is not necessarily equal to $L_2 \cdot L_1$).

Definition 0.8 (Kleene star) If $L \subseteq \Sigma^*$, then the *Kleene star* of L is the language L^* defined inductively by these rules:

$$\text{*Empty}\ \frac{}{\varepsilon \in L^*} \qquad \text{*Cat}\ \frac{w \in L \quad w \cdot u \in L^*}{w \cdot u \in L^*}$$

The following key property of the Kleene star is an easy consequence of this definition.

Theorem 0.9 *For any language L, we have $L^* = \{\varepsilon\} \cup L \cdot L^*$.*

Proof. We show both $L^* \subseteq \{\varepsilon\} \cup L \cdot L^*$ and $\{\varepsilon\} \cup L \cdot L^* \subseteq L^*$.

First, suppose that $v \in L^*$. By inversion on the derivation of $v \in L^*$, we have two cases:

Case *Empty: $v = \varepsilon$. Then $v \in \{\varepsilon\}$ and thus $v \in \{\varepsilon\} \cup L \cdot L^*$.

Case *Cat: $v = w \cdot u$ where $w \in L$ and $u \in L^*$. Then $v \in L \cdot L^*$ by the definition of concatenation and thus $v \in \{\varepsilon\} \cup L \cdot L^*$.

In either case, $v \in \{\varepsilon\} \cup L \cdot L^*$, and so we've shown $L^* \subseteq \{\varepsilon\} \cup L \cdot L^*$.

In the other direction, suppose $v \in \{\varepsilon\} \cup L \cdot L^*$. Then either $v = \varepsilon$, in which case $v \in L^*$ by (*Empty) or $v = w \cdot u$ where $w \in L$ and $u \in L^*$, in which case $v \in L^*$ by (*Cat). It follows that $\{\varepsilon\} \cup L \cdot L^* \subseteq L^*$. ⊣

In fact, we can extend this theorem further. Continuing the multiplication analogy, let $L^n = L \cdot L \cdots L$ (n times). That is,

$$L^0 = \mathbf{1} \quad \text{(i.e., } \{\varepsilon\}\text{)}$$
$$L^{n+1} = L \cdot L^n.$$

Then, for any n, we have

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup \cdots \cup L^n \cdot L^*,$$

as well as the infinite version:

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots = \bigcup_{i=0}^{\infty} L^i.$$

We won't use these characterizations of L^* in what follows, but you can try proving them as exercises.

The following theorem lists some other properites of the Kleene star:

Theorem 0.10 *(Properties of the Kleene star)*

1. $\varnothing^* = \{\varepsilon\}$ *(i.e.,* $0^* = 1$*)*
2. $\{\varepsilon\}^* = \{\varepsilon\}$ *(i.e.,* $1^* = 1$*)*
3. $(\Sigma)^* = \Sigma^*$*, justifying our use of* Σ^* *for the set of all strings on* Σ
4. $L \subseteq L^*$
5. $L_1 \subseteq L_2$ *implies* $L_1^* \subseteq L_2^*$
6. $L^* \cdot L^* \subseteq L^*$*, i.e.,* L^* *is closed under concatenation*
7. $(L^*)^* \subseteq L^*$*, i.e.,* L^* *is closed under Kleene star*

Moreover, the subset relationships in (6) and (7) can be replaced by equalities.

Proof. I'll prove (6) and leave the proofs of the rest as exercises.

By the observation following Definition 0.6 above, it will suffice to show that $w_1 \in L^*$ and $w_2 \in L^*$ imply $w_1 \cdot w_2 \in L^*$, which I'll do for an arbitrary $w_2 \in L^*$ by induction on the derivation of $w_1 \in L^*$.

Case *Empty: $w_1 = \varepsilon$. Then $w_1 \cdot w_2 = w_2$ by (CatEmpty), which is in L^* by assumption.

Case *Cat: $w_1 = w_1' \cdot u$ where $w_1' \in L$ and $u \in L^*$. Then

$$w_1 \cdot w_2 = (w_1' \cdot u) \cdot w_2 = w_1' \cdot (u \cdot w_2)$$

by Theorem 0.5.2. But by the IH on $u \in L^*$, we have $u \cdot w_2 \in L^*$, and so $w_1' \cdot (u \cdot w_2) \in L^*$ by *Cat, and therefore $w_1' \cdot (u \cdot w_2) = (w_1' \cdot u) \cdot w_2 = w_1 \cdot w_2$ again by Theorem 0.5.2.

To justify the statement that the subset relationship in (6) can be replaced by an equality, I need to prove the converse inclusion: $L^* \subseteq L^* \cdot L^*$. But if $w \in L^*$, then since $\varepsilon \in L^*$ by *Empty, we have $w = \varepsilon \cdot w \in L^* \cdot L^*$. ⊣

The conditions $\varepsilon \in L^*$, $L \subseteq L^*$, and $L^* \cdot L^* \subseteq L^*$ from *Empty and Theorem 0.10 can be used to give an alternative inductive definition of L^*:

Theorem 0.11 *(Alternative definition of Kleene star) L^* can be equivalently defined by the rules*

$$\frac{}{\varepsilon \in L^*} \qquad \frac{w \in L}{w \in L^*} \qquad \frac{w \in L^* \quad u \in L^*}{w \cdot u \in L^*}$$

i.e., it is the smallest set containing ε (as an element) and L (as a subset) and closed under concatenation.

Proof. Let's number these rules 1, 2, and 3, and for the purposes of this proof use L^{*1} and L^{*2} to stand for the sets defined by the rules in Definition 0.8 and the present theorem, respectively. I'll first show that $s \in L^{*1}$ implies $s \in L^{*2}$ for every $s \in \Sigma^*$ by induction on the derivation of $s \in L^{*1}$:

Case *Empty: $s = \varepsilon$. Then $s \in L^{*2}$ by rule 1.

Case *Cat: $s = w \cdot u$ where $w \in L$ and $u \in L^{*1}$. By the IH, $u \in L^{*2}$, and since $w \in L^{*2}$ by rule 2, we have $w \cdot u \in L^{*2}$ by rule 3.

Second, I'll show the converse, that $t \in L^{*2}$ implies $t \in L^{*1}$ for every $t \in \Sigma^*$, by induction on the derivation of $t \in L^{*2}$:

Case 1: $t = \varepsilon$. Then $t \in L^{*1}$ by *Empty.

Case 2: $t \in L$. Since $\varepsilon \in L^{*1}$ by *Empty and $t = t \cdot \varepsilon$ by Theorem 0.5, we have $t \in L^{*1}$ by *Cat.

Case 3: $t = w \cdot u$ where $w \in L^{*2}$ and $u \in L^{*2}$. By the IH, both $w \in L^{*1}$ and $u \in L^{*1}$, and so $t \in L^{*1}$ by Theorem 0.10.6. ⊣

Let me finish this section with yet another way of defining the Kleene star operation that makes it clear that L^* is the collection of all possible concatenations of elements of L. My notation is inspired by Haskell.

Given a set L, the set $[L]$ of *lists over* L can be defined inductively by the rules

$$\frac{}{[\,] \in [L]} \qquad \frac{w \in L \quad l \in [L]}{w : l \in [L]}$$

If L is a language, we can define an operation $\mathsf{concat} : [L] \to \Sigma^*$ by

$$\mathsf{concat}\ [\,] = \varepsilon \qquad \text{(ConcatEmpty)}$$

$$\mathsf{concat}\ (w : l) = w \cdot \mathsf{concat}\ l \qquad \text{(ConcatCons)}$$

Theorem 0.12 *(Yet another definition of Kleene star) L^* can be equivalently defined by the equation*

$$L^* = \{\mathsf{concat}\ l \mid l \in [L]\}.$$

Proof. Let's number the rules defining $[L]$ as 1 and 2. I'll first show by induction on the derivation of $s \in L^*$ that $s = \mathsf{concat}\ l$ for some $l \in [L]$. So, if $s = \varepsilon$, then we can take $l = [\,]$, since $l \in [L]$ by 1 and $\mathsf{concat}\ l = \varepsilon$ by (ConcatEmpty). And if $s = w \cdot u$ where $w \in L$ and $u \in L^*$, then by the IH, $u = \mathsf{concat}\ l'$ for some $l' \in [L]$, so we can take $l = w : l'$, and get $l \in [L]$ by 2 and $\mathsf{concat}\ l = s$ by (ConcatCons).

In the reverse direction, I'll show by induction on the derivation of $l \in [L]$ that $\mathsf{concat}\ l \in L^*$. So, if $l = [\,]$, then $\mathsf{concat}\ l = \varepsilon$ by (ConcatEmpty), which is in L^* by (*Empty). And if $l = w : l'$ where $w \in L$ and $l' \in [L]$, then $\mathsf{concat}\ l = w \cdot \mathsf{concat}\ l'$ by (ConcatCons), and since $\mathsf{concat}\ l' \in L^*$ by the IH, we have $\mathsf{concat}\ l \in L^*$ by (*Cat). ⊣

Regular languages and regular expressions

In the introduction, I defined regular languages as those accepted by DFAs. Here, I am going to take an indirect approach: I am going to give an inductive definition of what it means to be a "regular language" and then *prove*, in later sections, that these so-called regular languages are precisely the languages accepted by DFAs and thus are deserving of the name. The end point is the same, but this will allow me to do some preliminary study of regular languages before formally introducing DFAs and developing the tools necessary for the proof.

Regular languages

Definition 0.13 (**REG**) We define the class **REG** of *regular languages* by the following rules:

$$\textsc{RegEmpty}\ \frac{}{\varnothing \in \mathsf{REG}} \qquad \textsc{RegLetter}\ \frac{a \in \Sigma}{\{a\} \in \mathsf{REG}}$$

$$\textsc{RegUnion}\ \frac{L_1 \in \mathsf{REG} \quad L_2 \in \mathsf{REG}}{L_1 \cup L_2 \in \mathsf{REG}}$$

$$\textsc{RegCat}\ \frac{L_1 \in \mathsf{REG} \quad L_2 \in \mathsf{REG}}{L_1 \cdot L_2 \in \mathsf{REG}} \qquad \textsc{RegStar}\ \frac{L \in \mathsf{REG}}{L^* \in \mathsf{REG}}$$

Note that these rules define a *set* of languages, i.e., $\mathsf{REG} \subseteq \mathsf{Pow}(\Sigma^*)$ or, equivalently, $\mathsf{REG} \in \mathsf{Pow}(\mathsf{Pow}(\Sigma^*))$. It is the smallest set of languages that contains the empty and single-letter languages and is closed under union, concatenation, and Kleene star.

Let's play around with this definition a bit to get a sense of what's here. As a start, we can prove that every finite language is regular:

Theorem 0.14 *(Finite languages are regular) If $L \subseteq \Sigma^*$ is finite, then $L \in \mathsf{REG}$.*

The proof is to observe that, since the one-letter languages $\{a\}$ ($a \in \Sigma$) are all regular by (RegLetter) and regular languages are closed under concatenation by (RegCat), it follows that the *one-string* languages $\{w\}$ ($w \in \Sigma^*$) are also regular. And since every finite set is a union of one-element sets, we get that every finite language $F \subseteq \Sigma^*$ is regular.

Of course, regular languages can be infinite, too: if $\mathtt{a} \in \Sigma$, then by using (RegLetter) and (RegStar), we see that

$$\{\mathtt{a}\}^* = \{\varepsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \ldots\}$$

is regular. Finally, let's observe that, because of Theorem 0.14 and (REGUNION), any *finite modification* of a regular language is regular, where by finite modification I mean adding an arbitrary finite set of strings. (We'll see later that we can also *subtract* a finite set of strings and still have a regular language.)

Regular expressions

Regular expressions are just *names* of regular sets. Since, by Definition 0.13, regular sets are constructed using certain fixed operations, we can create a term language with constructors for each of these operations, and the terms will then correspond to the regular sets. Let's formalize this idea.

Let **RE** be the set of terms generated by this syntactic specification:

$$r \in \mathbf{RE} \quad ::= \quad 0 \mid \mathtt{a} \mid r_1 + r_2 \mid r_1 r_2 \mid r^* \qquad (\mathtt{a} \in \Sigma)$$

In writing such terms, it will be convenient to omit parentheses by declaring that the star operation has the highest precedence, followed by concatenation, and finally plus, and that both plus and concatenation associate to the right. Thus, the regular expression

$$\mathtt{b} + \mathtt{abc} + \mathtt{bc}^* \quad \text{is really} \quad (\mathtt{b} + (((\mathtt{a})((\mathtt{b})(\mathtt{c}))) + ((\mathtt{b})((\mathtt{c})^*)))),$$

when a pair of parentheses is added around the result of every constructor.

If r is a regular expression, let us write $[\![r]\!]$ for the regular set named by r. This can be defined recursively as follows:

$$\begin{aligned} [\![0]\!] &= \varnothing \\ [\![\mathtt{a}]\!] &= \{\mathtt{a}\} \\ [\![r_1 + r_2]\!] &= [\![r_1]\!] \cup [\![r_2]\!] \\ [\![r_1 r_2]\!] &= [\![r_1]\!] \cdot [\![r_2]\!] \\ [\![r^*]\!] &= [\![r]\!]^*. \end{aligned}$$

Of course, because of the nature of the regular operators, many different regular expressions can denote the same regular set. For example, both $\mathtt{a} + \mathtt{b}$ and $\mathtt{b} + \mathtt{a}$ denote the regular set $\{\mathtt{a}, \mathtt{b}\}$; both $\mathtt{a} + (\mathtt{b} + \mathtt{c})$ and $(\mathtt{a} + \mathtt{b}) + \mathtt{c}$ denote the regular set $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$; and, for any regular expression r, the regular expressions r^*, $(r^*)^*$, and $r^* r^*$, all denote the regular set $[\![r]\!]^*$, as a consequence of Theorem 0.10.

If w is a string and r is a regular expression, then we say that w *matches* r if $w \in [\![r]\!]$.

Examples. I'm going to give a few easier examples of regular expressions over the two-letter alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and then leave the harder ones as exercises for you!

1. The regular expression $r = (\mathtt{a} + \mathtt{b})^*$ matches all strings on Σ, i.e., every string of a's and b's, of any length, including the empty string.

2. Both regular expressions $r_1 = (\mathtt{a} + \mathtt{b})(\mathtt{a} + \mathtt{b})^*$ and $r_2 = (\mathtt{a} + \mathtt{b})^*(\mathtt{a} + \mathtt{b})$ match all non-empty strings, since the $(\mathtt{a} + \mathtt{b})$ term matches exactly one character.

3. All three regular expressions $r_1 = \mathtt{b}^*\mathtt{a}(\mathtt{a} + \mathtt{b})^*$, $r_2 = (\mathtt{a} + \mathtt{b})^*\mathtt{a}(\mathtt{a} + \mathtt{b})^*$ and $r_3 = (\mathtt{a} + \mathtt{b})^*\mathtt{a}\mathtt{b}^*$ match all strings containing at least one `a`: r_1 singles out the first `a`, r_2 singles out any `a`, and r_3 singles out the last `a`.

4. The regular expression $r = \mathtt{a}^* + \mathtt{a}^*\mathtt{b}\mathtt{a}^*$ matches all strings with at most one `b`: $\mathtt{a}^*$ matches strings with no `b`'s, and $\mathtt{a}^*\mathtt{b}\mathtt{a}^*$ matches strings with exactly one `b`.

Exercises. Now, try your hand at constructing regular expressions for each of the following (more difficult) languages, still over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$:

1. all strings with at least one `a` and at least one `b`

2. all strings in which every `a` is immediately followed by `bb`

3. all strings with an even number of `a`'s

4. all strings with no instance of `bbb` (as a substring)

5. all strings with every instance of `aa` coming before every instance of `bb`

6. all strings with no instance of `aba`

7. all strings with an even number of `a`'s and an odd number of `b`'s.

Finite State Machines

We now come to the problem of turning our previous informal description of a DFA into a formal definition. Recall from the Introduction that a DFA

- reads characters one at a time from left to right, indicating after each character whether it has accepted the string read so far,
- can only be in one of a fixed, finite number of states, and
- is deterministic (i.e., always starts in the same state and behaves the same way given the same input).

We can represent the states of a DFA as a finite set Q, and let $s \in Q$ be the state in which the machine starts (its *start state*). Since the machine is deterministic, and the only "memory" available to it is its set of states, all that matters in determining the machine's next state is the state that it is currently in and the character it just read. It follows that the machine's transition behavior is completely described by a function $\delta : Q \times \Sigma \to Q$, so that if the machine is in state $q \in Q$ and reads the character $x \in \Sigma$, it moves to the state $\delta(q, x)$.

That leaves the question of acceptance: how does the machine indicate it has accepted the string read so far? Its decision to accept a string or not must be completely determined by the state it's in after reading the last character, so that leaves us really only one choice: we designate some subset $F \subseteq Q$ of the states as *accepting states* and say that if the machine enters an accepting state then it has accepted the string it has read so far. With these preliminaries, we can now give the formal definition.

Definition of a DFA

Definition 0.15 (Finite State Machine) A *Finite State Machine* is a tuple (Q, s, F, δ), where

- Q is a finite set (the *states*)
- $s \in Q$ (the *initial state*)
- $F \subseteq Q$ (the *final states*)
- $\delta : Q \times \Sigma \to \Sigma$ (the *transition function*)

We denote by DFA the set of all finite state machines.

We want to have no limitation on what Q can be, other than that it is finite, so that we have maximum flexibility in constructing machines. A much more restrictive alternative would have been to replace Q with a single number N and call the states $\{0, 1, 2, \ldots, N-1\}$. Although this would work, and would standardize machines more, it would make it much harder to construct new machines from old ones.

Language Accepted by a DFA

Of course, the main purpose of a DFA is to accept a particular language, so we have to give a formal definition of the language accepted by a DFA $M = (Q, s, F, \delta)$ in terms of its components Q, s, F, and δ. We will denote this language by $L(M)$, which you can read as "the language of M." Actually, I'm going to give two such definitions, each of which proceeds by recursion on the input string, and then prove (by induction on the input string) that they are equivalent.

Definition 1. For this definition we extend the function δ, which tells us what M does on each input symbol, to a function δ^*, which tells us what M does on each input *string*. Since a string is just a sequence of input symbols, this can be achieved by recursion on the input string:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, aw) = \delta^*(\delta(q, a), w)$$

That is, after "reading" the empty string, we are still in the same state, and after reading aw, we are in the state we get to by first making the transition from q to $\delta(q, a)$ on a, and then reading the rest of the string w from there.

Now, using δ^* we can define $L(M)$:

$$w \in L(M) \quad \text{iff} \quad \delta^*(s, w) \in F.$$

In English: a string w is accepted by the DFA M if, when we start the machine in state s and read the string w, we end up in a final state.

Definition 2. For this defintion, we generalize $L(M)$ to $L_q(M)$, which is the language accepted by M when we start the machine in state q rather than s. The judgment $w \in L_q(M)$ can be defined, simultaneously for every $q \in Q$, by these rules:

$$\textsc{AcceptEmpty}\frac{q \in F}{\varepsilon \in L_q(M)} \qquad \textsc{AcceptCons}\frac{w \in L_{\delta(q,a)}(M)}{aw \in L_q(M)}$$

In English: we accept the empty string starting in state q if q is already a final state, and we accept the string aw starting in state q if we accept the string w starting in state $\delta(q, a)$, which is where we are after making the transition from q on symbol a.

Using this judgment, we can give our second definition of $L(M)$:

$$w \in L(M) \quad \text{iff} \quad w \in L_s(M).$$

The equivalence. How do we prove the equivalence of these two definitions? Clearly, given the two ways we've defined $L(M)$, this will amount to showing

$$\delta^*(s, w) \in F \quad \text{iff} \quad w \in L_s(M),$$

for all w, and we would be tempted to prove this by induction on w, since both of the definitions proceeded by recursion on w. However, this won't work, because both definitions involve s being replaced by other states as we recurse down the input string w, and so this equivalence is too weak to use as an IH. What we need is a stronger IH that quantifies over *all* states:

Theorem 0.16 *Suppose $M = (Q, s, F, \delta)$ is a DFA, and δ^* and $L_q(M)$ are defined as above. Then for any string $w \in \Sigma^*$, we have*

$$\textit{for all } q \in Q,\ \delta^*(q, w) \in F \textit{ iff } w \in L_q(M).$$

Proof. By induction on w (where we note that the quantification over states is now included in the IH).

Case Empty: $w = \varepsilon$. Let q be arbitrary. Then, by the definitions of δ^* and $L_q(M)$, we have $\delta^*(q, w) \in F$ iff $q \in F$ iff $w \in L_q(M)$, as required.

Case CONS: $w = aw'$, where $a \in \Sigma$ and $w' \in \Sigma^*$. Let q be arbitrary. Then,

$$\begin{aligned} \delta^*(q, aw') \in F \quad &\text{iff} \quad \delta^*(\delta(q,a), w') \in F && \text{(by def of } \delta^*) \\ &\text{iff} \quad w' \in L_{\delta(q,a)}(M) && \text{(by the IH on state } \delta(q,a)) \\ &\text{iff} \quad aw' \in L_q(M), && \text{(by def of } L_q(M)) \end{aligned}$$

as required (and note how the stronger IH lets us use the equivalence at any state, in this case $\delta(q,a)$, even though we have to prove it for q). ⊣

The equivalence of the two definitions of $L(M)$ now follows easily from this Theorem by choosing q to be s. The practical implication of this equivalence is that we are free to use either definition where it is convenient, although it actually seems in what follows that I always use Definition 1, and so I will take this definition as the official one.

The construction of the language $L(M)$ from a DFA M is of course functional, so that we actually have $L : \mathsf{DFA} \to \mathsf{Pow}(\Sigma^*)$. But will see in Section below that $L(M)$ is, in fact, regular for every $M \in \mathsf{DFA}$, meaning that our function is actually $L : \mathsf{DFA} \to \mathsf{REG}$.

Representing DFAs Visually

In Section , we represented (or named) regular languages using regular expressions, and we would like to do something similar with DFAs, but we quickly realize that the amount of information that goes into a description of a DFA is too much for a useful textual description. Fortunately, there is a nice graphical description that will do the job.

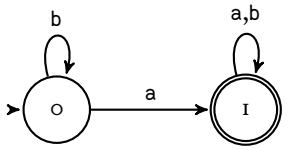
We will represent a DFA $M = (Q, s, F, \delta)$ graphically by drawing a circle for each state; drawing an arrow connecting two circles, labeled with a list (or set) of input symbols, if there are transitions from the source state to the destination state for each of input symbols in the label; drawing an arrow from the word "start" into the initial state; and making the final state(s) double circles.

For example, consider the DFA over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ with

- $Q = \{0, 1\}$
- $s = 0$
- $F = \{1\}$
- $\delta = \{(0, \mathtt{a}, 1), (0, \mathtt{b}, 0), (1, \mathtt{a}, 1), (1, \mathtt{b}, 1)\}$,

where I've given δ by its "graph;" thus, $\delta(0, \mathtt{a}) = 1$, $\delta(0, \mathtt{b}) = 0$, $\delta(1, \mathtt{a}) = 1$, and $\delta(1, \mathtt{b}) = 1$. This DFA can be represented graphically as follows:

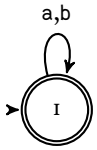
M:



Examples and Exercises

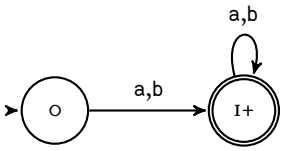
We will now revisit the examples and exercises from Section , this time constructing DFAs that accept the given languages. As before, I will do the easier ones and leave the more difficult ones for you! (Part of the fun of learning how to design DFAs is figuring out the tricks and techniques yourself, but I'll give you a few hints to get you started.) Again, we are keeping to a two-letter alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

1. The language consisting of all strings can be recognized by a one-state DFA:



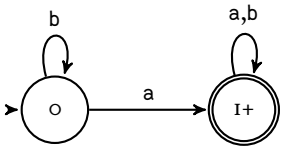
   Note, by the way, that if the single state in this machine were not a final state, then the machine would accept the empty language—consider that a bonus example!

2. The language consisting of all nonempty strings can be recognized by a machine that adds a new initial state to the previous machine:



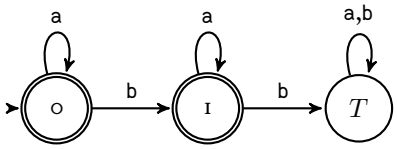
   Here, the names I have chosen for the states are suggestive: 0 is the state where we have (so far) read 0 characters, and 1+ is the state where we have read 1 or more characters.

3. The language consisting of all strings with at least one a is recognized by the example machine from the previous subsection, which I'll repeat here for convenience (with a new name for state 1):



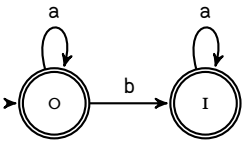
   State 0 of this machine is the state where we have read 0 a's, and state 1+ is the state where we have read one or more a's; note that reading a b doesn't change the number of a's we have read, so these transitions just loop back to the same state.

4. Finally, the language consisting of all strings with at most one b can be recognized by this three-state DFA:



Here, state 0 is the state where we have read 0 b's, state 1 is the state where we have read 1 b, and state T is a "trap state," i.e., a state that is not final but to which all transitions lead back. Both 0 and 1 are final states, because we want to accept strings that have 0 b's or exactly 1 b. However, once we read a second b, we will no longer accept the string, whatever the remaining characters may be, and so a trap state is appropriate.

Trap states are both useful and common, and so we will adopt a convention when drawing machines with trap states that will keep our diagrams simpler: if, in a DFA diagram, there is at least one state and input symbol for which no transition is indicated, then the machine is assumed to have a single trap state, and every missing transition is taken to be a transition to that state. Thus, we could simplify the previous machine as follows:



Now, with these examples in hand, go back to the exercises in Section and construct DFAs for each of those languages. In some cases, you will be able to use the trap-state convension to keep your machines simpler.

Regular Languages are Accepted by DFAs

The goal of this section is to prove one half of the equivalence of DFAs and regular languages:

Theorem 0.17 *For every regular language R, there exists a finite state machine M such that $R = L(M)$.*

Our strategy is simple, even if the individual steps are more complicated. The class of regular languages has an inductive definition (0.13), according to which the empty language and, for every $a \in \Sigma$, the single-letter language, $\{a\}$, are regular; and if L_1 and L_2 are regular, then $L_1 \cup L_2$, $L_1 \cdot L_2$, and L_1^* are regular. Moreover, we have a system of names for these regular languages, which we called regular expressions (the set of which was denoted **RE**), with a constructor for each of these base languages and regular operators; if $r \in \mathsf{RE}$, then $[\![r]\!] \in \mathsf{REG}$ was the regular language named by r. Therefore, to show that every regular language is accepted by some DFA, it will be enough to

- construct a machine $M(0)$ that accepts the (empty) language, $\varnothing$;

- construct, for every $a \in \Sigma$, a machine $M(a)$ that accepts the (single-letter) language, $\{a\}$;
- given machines M_1 and M_2 that accept the languages L_1 and L_2, construct a machine $M_1 \cup M_2$ that accepts the language $L_1 \cup L_2$;
- given machines M_1 and M_2 that accept the languages L_1 and L_2, construct a machine $M_1 \cdot M_2$ that accepts the language $L_1 \cdot L_2$; and
- given a machine M that accepts the language L, construct a machine M^* that accepts the language L^*.

This collection of constructions can then be used to give a recursive definition of a function $M : \mathsf{RE} \to \mathsf{DFA}$ that satisfies the condition $\forall r \in \mathsf{RE}\ L(M(r))) = [\![r]\!]$. Since every regular language, R, satisfies $[\![r]\!] = R$ for its name, r, the machine $M(r)$ will be therefore be the required machine recognizing R.

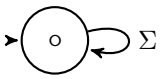
Each one of the above bulleted constructions is the subject of its own subsection below.

Empty

I've already given a machine accepting the empty language—it was the "bonus" machine from Example .1: let $M(0)$ be the machine with

- $Q = \{0\}$
- $s = 0$
- $F = \varnothing$
- $\delta(0, a) = 0$.

This machine can be pictured as follows:



Let me record the obvious:

Theorem 0.18 $L(M(0)) = \varnothing$.

Proof. By the definition of $L(M)$, $w \in L(M(0))$ iff $\delta^*(0, w) \in \varnothing$. But the latter is false for every w, and so $L(M(0))$ is empty. ⊣

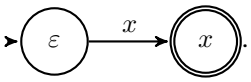
Letter

For any $x \in \Sigma$, let $M(x)$ be the machine with

- $Q = \{\varepsilon, x, T\}$
- $s = \varepsilon$

- $F = \{x\}$
- $\delta(q, a) = \begin{cases} x, & \text{if } q = \varepsilon \text{ and } a = x; \\ T, & \text{otherwise.} \end{cases}$

This machine can be pictured as follows, where I am using the trap-state convention to avoid having to draw the trap state, T:



I've chosen the names of the (non-trap) states to indicate the strings that must have already been read by the machine if it ends up in that particular state.

Now, it's probably also obvious to you that this machine accepts exactly the language $\{x\}$. Nevertheless, I will give a formal proof of this, so that you can see how it is done. I'll start, as I will for the other constructions to come, with a characterization of δ^* for this machine:

Lemma 0.19 *For all $w \in \Sigma^*$,*

1. $\delta^*(T, w) = T$
2. $\delta^*(x, w) = \begin{cases} x, & \textit{if } w = \varepsilon; \\ T, & \textit{otherwise} \end{cases}$
3. $\delta^*(\varepsilon, w) = \begin{cases} \varepsilon, & \textit{if } w = \varepsilon; \\ x, & \textit{if } w = x; \\ T, & \textit{otherwise} \end{cases}$

Proof. I'll prove 1 by induction on w. For $w = \varepsilon$, we have $\delta^*(T, \varepsilon) = T$ by definition of δ^*. For $w = aw'$, we have, by the definitions of δ^* and δ and the IH,

$$\delta^*(T, aw') = \delta^*(\delta(T, a), w') = \delta^*(T, w') = T.$$

I'll prove 2 by inversion on w, with two cases. For $w = \varepsilon$, we have $\delta^*(x, \varepsilon) = x$ by definition of δ^*. For $w = aw'$, we have, by the definitions of δ^* and δ and part 1,

$$\delta^*(x, aw') = \delta^*(\delta(x, a), w') = \delta^*(T, w') = T.$$

I'll prove 3 also by inversion on w, with four cases. For $w = \varepsilon$, we have $\delta^*(\varepsilon, w) = \varepsilon$ by definition of δ^*. For $w = x$, we have, by the definitions of δ^* and δ,

$$\delta^*(\varepsilon, x) = \delta^*(\delta(\varepsilon, x), \varepsilon) = \delta^*(x, \varepsilon) = x.$$

For $w = xw'$ with $w' \neq \varepsilon$, we have, by the definitions of δ^* and δ and part 2,

$$\delta^*(\varepsilon, xw') = \delta^*(\delta(\varepsilon, x), w') = \delta^*(x, w') = T.$$

Finally, for $w = aw'$ with $a \neq x$, we have, by the definitions of δ^* and δ and part 1,

$$\delta^*(\varepsilon, aw') = \delta^*(\delta(\varepsilon, a), w') = \delta^*(T, w') = T.$$

⊣

Theorem 0.20 $L(M(a)) = \{a\}$.

Proof. By the definition of $L(M)$, $w \in L(M(a))$ iff $\delta^*(\varepsilon, w) \in \{a\}$. But part 3 of the Lemma shows that the latter occurs exactly when $w = a$. ⊣

Union

In this case, we are given two machines

$$M_1 = (Q_1, s_1, F_1, \delta_1) \qquad \text{and} \qquad M_2 = (Q_2, s_2, F_2, \delta_2),$$

and need to construct a machine $M_1 \cup M_2$ with $L(M_1 \cup M_2) = L(M_1) \cup L(M_2)$. Here is the idea. Given an input string w, we are going to start both machines M_1 and M_2 in their respective start states simultaneously reading w. As they each make their respective transitions, we will record the simultaneous state of both machines as a pair (q_1, q_2), where q_1 is the state of machine M_1 and q_2 is the state of machine M_2. Since we want $M_1 \cup M_2$ to accept a string if it is accepted by *either* M_1 or M_2 (or both), we can simply declare the final states of $M_1 \cup M_2$ to be the states (q_1, q_2) where $q_1 \in F_1$ or $q_2 \in F_2$.

Here, then, is the formal construction. Let $M_1 \cup M_2$ be the machine with

- $Q = Q_1 \times Q_2$
- $s = (s_1, s_2)$
- $F = \{(q_1, q_2) \mid q_1 \in F_1 \vee q_2 \in F_2\} = (F_1 \times Q_2) \cup (Q_1 \times F_2)$
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$.

To prove that this construction works, we again give the appropriate characterization of δ^* for this machine:

Lemma 0.21 *For all* $(q_1, q_2) \in Q$ *and* $w \in \Sigma^*$,

$$\delta^*((q_1, q_2), w) = (\delta_1^*(q_1, w), \delta_2^*(q_2, w)).$$

Proof. By induction on w. For $w = \varepsilon$, we have

$$\begin{aligned} \delta^*((q_1, q_2), \varepsilon) &= (q_1, q_2) && \text{(def of } \delta^*) \\ &= (\delta_1^*(q_1, \varepsilon), \delta_2^*(q_2, \varepsilon)). && \text{(defs of } \delta_1^* \text{ and } \delta_2^*) \end{aligned}$$

For $w = aw'$, we have

$$\begin{aligned}
\delta^*((q_1, q_2), aw') &= \delta^*\big(\delta\big((q_1, q_2), a\big), w'\big) && \text{(def of } \delta^*\text{)} \\
&= \delta^*\big(\big(\delta_1(q_1, a), \delta_2(q_2, a)\big), w'\big) && \text{(def of } \delta\text{)} \\
&= \big(\delta_1^*(\delta_1(q_1, a), w'), \delta_2^*(\delta_2(q_2, a), w')\big) && \text{(IH)} \\
&= \big(\delta_1^*(q_1, aw'), \delta_2^*(q_2, aw')\big). && \text{(defs of } \delta_1^* \text{ and } \delta_2^*\text{)}
\end{aligned}$$

as required. ⊣

Theorem 0.22 $L(M_1 \cup M_2) = L(M_1) \cup L(M_2)$.

Proof. Let $w \in \Sigma^*$ be arbitrary. Then

$$\begin{aligned}
w \in L(M_1 \cup M_2) \quad \text{iff} \quad & \delta^*((s_1, s_2), w) \in \{(q_1, q_2) \mid q_1 \in F_1 \vee q_2 \in F_2\} \\
& \qquad \text{(def of machine and } L\text{)} \\
\text{iff} \quad & (\delta_1^*(s_1, w), \delta_2^*(s_2, w)) \in \{(q_1, q_2) \mid q_1 \in F_1 \vee q_2 \in F_2\} \\
& \qquad \text{(Lemma)} \\
\text{iff} \quad & \delta_1^*(s_1, w) \in F_1 \vee \delta_2^*(s_2, w) \in F_2 \\
\text{iff} \quad & w \in L(M_1) \vee w \in L(M_2) \qquad \text{(def of } L\text{)} \\
\text{iff} \quad & w \in L(M_1) \cup L(M_2),
\end{aligned}$$

as required. ⊣

Exercise. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Draw the machine $M(\mathtt{a}) \cup M(\mathtt{b})$, first without using the trap-state convention, and then again using it. (Hint: the first machine will have 9 states, 5 of which are final, and 14 arrows; the second machine will have 8 states and 6 arrows.) What's the smallest machine you can draw that accepts the same language, i.e., $\{\mathtt{a}, \mathtt{b}\}$? What's going on with the extra states in $M(\mathtt{a}) \cup M(\mathtt{b})$? (The answer will be given in Section .)

Concatenation

Similarly to the union, we are given two machines M_1 and M_2 and need to construct a machine $M_1 \cdot M_2$ such that $L(M_1 \cdot M_2) = L(M_1) \cdot L(M_2)$. This construction and its analysis are a bit tricky, however, so let's first get a better idea of what's required. By definition of concatenation, $w \in L(M_1) \cdot L(M_2)$ iff we can write $w = w_1 \cdot w_2$, where $w_1 \in L(M_1)$ and $w_2 \in L(M_2)$. Now if there were just one possible way to divide w into w_1 and w_2 such that $w_1 \in L(M_1)$, then we could proceed as follows:

- Start M_1 reading w.
- If M_1 enters a final state after reading w_1 (with w_2 still to read), then abandon machine M_1 and start M_2 in its start state, s_2, reading w_2.

- If M_2 is in a final state after reading w_2, then accept, otherwise reject.

The problem is that there might be several ways to divide w as $w_1 \cdot w_2$ with $w_1 \in L(M_1)$, only one of which has $w_2 \in L(M_2)$. If we commit to running M_2 on w_2 as soon as we find the *first* w_1 such that $w = w_1 \cdot w_2$ and $w_1 \in L(M_1)$, it may be that M_2 rejects *that* w_2, whereas a different division of w into $w_1 \cdot w_2$ would have succeeded.

What to do? The solution is a combination of two insights. First, in order to keep all of our options open, we won't abandon machine M_1 when it gets into a final state, but keep it running simultaneously with M_2, which we have just started in its start state. Then, if M_1 gets into a final state again later on, we'll start *another copy* of M_2 in its start state, so that we'll have three machines running simultaneously. And, as M_1 might continue returning to a final state many more times while reading w, we might end up having many copies of M_2 running simultaneously, each in its own state. Finally, when we get to the end of the string w, we will say that this system of machines accepts w if *at least one* of the copies of M_2 is in a final state.

The problem with this idea, besides it being an accounting nightmare, is that there would be no limit to the number of copies of M_2 we'd have to keep track of simultaneously, and we only have available a fixed, finite amount of state that was independent of the length of the input string. This is where the second insight comes in. Since all we care about this set of copies of M_2 is that at least one of them gets into a final state after finishing w, all we really need to keep track of is the particular *subset* of states of M_2 that the system of machines is in, as an aggregate, at any one time—having more than one copy of M_2 in the same state doesn't make it any easier or harder to accept the string w, since they will all accept or reject together. And if Q_2 is a finite set, then $\mathsf{Pow}(Q_2)$ is also finite.

Thus, the states of the machine $M_1 \cdot M_2$ will be pairs (q_1, X_2), where $q_1 \in Q_1$ and $X_2 \subseteq Q_2$, with X_2 representing the aggregate of the states the copies of the machine M_2 are in at the moment. There is one catch, however: if $q_1 \in F_1$ then we should have $s_2 \in X_2$ as well, since we will be starting another copy of M_2 in its start state at the same time. This condition on states can insured by the following device: if X_2 occurs as the second component of a state (q_1, X_2), then we define

$$X_2^c = X_2 \cup \{s_2 \mid q_1 \in F_1\},$$

which "corrects" X_2 by adding s_2 to X_2 if the first component of the state is final and leaving it unchanged if not. Note that correction is *idempotent*, in the sense that correcting a pair that's already corrected doesn't result in any further changes: $(X_2^c)^c = X_2^c$.

In defining the final states, I will use the notation $X_2 \sqcap F_2$ to mean that the sets X_2 and F_2 *overlap*, i.e., that there exists $q_2 \in X_2 \cap F_2$ (or, written another way, $X_2 \cap F_2 \neq \varnothing$), meaning that at least one of the copies of M_2 is in a final state.

Finally, in defining the transition function, I will make use of the function $\hat{\delta}_2 : \mathsf{Pow}(Q_2) \times \Sigma \to \mathsf{Pow}(Q_2)$ defined by

$$\hat{\delta}_2(X_2, a) = \{\delta_2(q_2, a) \mid q_2 \in X_2\},$$

which are the states to which the various copies of M_2 transition on symbol a.

With these preliminaries out of the way, I can now give the formal construction: let $M_1 \cdot M_2$ be the machine with

- $Q = \{(q_1, X_2^c) \mid q_1 \in Q_1, X_2 \subseteq Q_2\}$
- $s = (s_1, \varnothing^c)$
- $F = \{(q_1, X_2) \in Q \mid X_2 \sqcap F_2\}$
- $\delta\big((q_1, X_2), a\big) = (\delta_1(q_1, a), \hat{\delta}_2(X_2, a)^c)$.

Note that, from the definition of s, we don't start any copies of M_2 at the beginning, *unless* s_1 is also a final state of M_1, in which case we are also starting one copy of M_2 at the same time. Also, we always correct the result of a transition, since $\delta_1(q_1, a)$ might be a final state of M_1. These corrections insure that $s \in Q$ and $\delta : Q \times \Sigma \to Q$, as required in the definition of a DFA.

Now, to prove the correctness of this construction, I'll again state and prove the appropriate characterization of δ^* for this machine, which in this case will make use of an auxiliary notion, called a *trace*, that will be useful also in the analysis of the construction of M^* in the next section. To motivate this notion, consider what happens when our machine is in state $(q_1, X_2) \in Q$ and we read the string w, arriving in state (q_1', X_2'): where do the elements of X_2' come from? There are two sources: (1) a copy of M_2 that was in state $q_2 \in X_2$ will transition on w to the state $\delta_2^*(q_2, w) \in X_2'$, and (2) M_1, reading w, will transition to the state $\delta_1^*(q_1, w)$, but along the way, it may enter a final state of M_1 after reading w_1, causing a copy of M_2 to start up on the string w_2, where $w = w_1 \cdot w_2$, which will then transition to $\delta_2^*(s_2, w_2) \in X_2'$.

Here, then, is the formal definition of a trace and the corresponding characterization of δ^* for our concatenation machine:

Definition 0.23 Suppose $(q_1, X_2) \in Q$ and $w \in \Sigma^*$. Then a *trace from* (q_1, X_2) *on* w is an object of one of the following two forms:

- $(q_2 \,|\, w)$, where $q_2 \in X_2$, or
- $(q_1 \,|\, w_1, w_2)$, where $w_1 \neq \varepsilon$ and $\delta_1^*(q_1, w_1) \in F_1$ and $w = w_1 \cdot w_2$.

We let $\mathsf{Tr}(q_1, X_2 \,|\, w)$ be the set of all traces from (q_1, X_2) on w and define a function $\mathsf{end} : \mathsf{Tr}(q_1, X_2 \,|\, w) \to Q_2$ by

$$\mathsf{end}(q_2 \,|\, w) = \delta_2^*(q_2, w)$$
$$\mathsf{end}(q_1 \,|\, w_1, w_2) = \delta_2^*(s_2, w_2),$$

which gives the *end-point* of a trace. Finally, we define the set

$$E((q_1, X_2), w) = \{\mathsf{end}(t) \mid t \in \mathsf{Tr}(q_1, X_2 \mid w)\}$$

of all end-points of traces from (q_1, X_2) on w.

Lemma 0.24 *For all $(q_1, X_2) \in Q$ and $w \in \Sigma^*$,*

$$\delta^*\big((q_1, X_2), w\big) = \big(\delta_1^*(q_1, w), E((q_1, X_2), w)\big).$$

Proof. Let $(q_1, X_2) \in Q$. I'll first establish the following recursion equations for the sets $E((q_1, X_2), w) \subseteq Q_2$:

$$E((q_1, X_2), \varepsilon) = X_2 \tag{1}$$

$$E((q_1, X_2), aw) = E\big((\delta_1(q_1, a), \hat{\delta}_2(X_2, a)^c), w\big). \tag{2}$$

The proof of (1) is summarized by the table,

$t \in \mathsf{Tr}(q_1, X_2 \mid \varepsilon)$	$q_2 \in X_2$
$(q_2 \mid \varepsilon)$	q_2

which I claim defines a total and onto relation (in fact, bijection) between elements $t \in \mathsf{Tr}(q_1, X_2, \varepsilon)$ and elements $q_2 \in X_2$, the connection being $\mathsf{end}(t) = q_2$. The existence of such a relation establishes (1).

Similarly, the proof of (2) is summarized by the table,

$t \in \mathsf{Tr}(q_1, X_2 \mid aw)$	$t' \in \mathsf{Tr}(\delta_1(q_1, a), \hat{\delta}_2(X_2, a)^c \mid w)$	condition
$(q_2 \mid aw)$	$(\delta_2(q_2, a) \mid w)$	
$(q_1 \mid aw_1', w_2)$	$(\delta_1(q_1, a) \mid w_1', w_2)$	$w_1' \neq \varepsilon$
$(q_1 \mid a, w)$	$(s_2 \mid w)$	$\delta_1(q_1, a) \in F_1$

which I claim defines three parts of a total and onto relation between elements $t \in \mathsf{Tr}(q_1, X_2 \mid aw)$ and elements $t' \in \mathsf{Tr}(\delta_1(q_1, a), \hat{\delta}_2(X_2, a)^c \mid w)$, under the indicated conditions, such that $\mathsf{end}(t) = \mathsf{end}(t')$. The existence of such a relation establishes (2). In both cases, I leave the straightforward checks to the reader.

With (1) and (2) established, the proof of the main lemma is now a straightforward induction on w. For $w = \varepsilon$, we have

$$\begin{aligned}
\delta^*((q_1, X_2), \varepsilon) &= (q_1, X_2) && \text{(def of } \delta^*\text{)} \\
&= \big(\delta_1^*(q_1, \varepsilon), E((q_1, X_2), \varepsilon)\big). && \text{(def of } \delta_1^* \text{ and (1))}
\end{aligned}$$

For $w = aw'$, we have

$$\begin{aligned}
\delta^*((q_1, X_2), aw') &= \delta^*(\delta((q_1, X_2), a), w') && \text{(def of } \delta^*\text{)} \\
&= \delta^*((\delta_1(q_1, a), \hat{\delta}_2(X_2, a)^c), w') && \text{(def of } \delta\text{)} \\
&= \big(\delta_1^*(\delta_1(q_1, a), w'), E((\delta_1(q_1, a), \hat{\delta}_2(X_2, a)^c), w')\big) && \text{(IH)} \\
&= (\delta_1^*(q_1, aw'), E((q_1, X_2), aw')). && \text{(def of } \delta_1^* \text{ and (2))}
\end{aligned}$$

completing the proof of the Lemma. ⊣

Theorem 0.25 $L(M_1 \cdot M_2) = L(M_1) \cdot L(M_2)$.

PROOF. Let $w \in \Sigma^*$ be arbitrary. Then

$$\begin{aligned} w \in L(M_1 \cdot M_2) \quad &\text{iff} \quad \delta^*((s_1, \varnothing^c), w) \in \{(q_1, X_2) \in Q \mid X_2 \sqcap F_2\} \\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{(def of machine and } L\text{)} \\ &\text{iff} \quad E((s_1, \varnothing^c), w) \sqcap F_2 \qquad\qquad\qquad\qquad \text{(Lemma)} \\ &\text{iff} \quad \exists t \in \mathsf{Tr}(s_1, \varnothing^c \mid w)\ \mathsf{end}(t) \in F_2. \qquad\qquad \text{(def of } E\text{)} \end{aligned}$$

Now, any such t is either $t = (s_2 \mid w)$ with $s_1 \in F_1$ and $\delta_2^*(s_2, w) \in F_2$, in which case the decomposition $w = \varepsilon \cdot w$ shows that $w \in L(M_1) \cdot L(M_2)$, or $t = (s_1 \mid w_1, w_2)$ with $\delta_1^*(s_1, w_1) \in F_1$ and $\delta_2^*(s_2, w_2) \in F_2$, in which case the decomposition $w = w_1 \cdot w_2$ shows that $w \in L(M_1) \cdot L(M_2)$. Conversely, if $w \in L(M_1) \cdot L(M_2)$, then $w = w_1 \cdot w_2$ for $w_1 \in L(M_1)$ and $w_2 \in L(M_2)$, i.e., $\delta_1^*(s_1, w_1) \in F_1$ and $\delta_2^*(s_2, w_2) \in F_2$, and so the trace $t = (s_1 \mid w_1, w_2)$ satisfies $t \in \mathsf{Tr}(s_1, \varnothing^c \mid w)$ and $\mathsf{end}(t) \in F_2$, showing, as above, that $w \in L(M_1 \cdot M_2)$ and completing the proof. ⊣

Exercise. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Draw the machine $M(\mathtt{a}) \cdot M(\mathtt{b})$. (Hint: it will have 20 states, 10 of which are final.) What's the smallest machine you can draw that accepts the same language, i.e., $\{\mathtt{ab}\}$? Again, what's going on with all these extra states? (Again, the answer will be given in Section .)

Star

For the final construction, we are given a machine $M_1 = (Q_1, s_1, F_1, \delta_1)$ and we need to construct a machine M_1^* such that $L(M_1^*) = L(M_1)^*$. The idea is similar to the concatenation machine, in that we will need to run multiple copies of M_1 simultaneously, so we will use subsets $X \subseteq Q_1$ to keep track of the aggregate set of states a collection of copies of M_1 are in at any given time, with the catch that, if $X \sqcap F_1$, then we require $s_1 \in X$; that is, when at least one copy of M_1 is in a final state, then a new copy of M_1 will be started simultaneously in its initial state, which we will enforce by defining, for $X \subseteq Q_1$,

$$X^c = X \cup \{s_1 \mid X \sqcap F_1\}.$$

A set of states is final if it includes at least one final state. One final twist: since we always have $\varepsilon \in M_1^*$, the start state of M_1^* must also be final, even if the start state of M_1 is not. We will achieve this by using $\varnothing$ as the start state of M_1^*, which is not otherwise used in the construction, and set the transitions out of this state to mirror the transitions of M_1 out of s_1.

Here, then is the formal construction. Let M_1^* be the machine with

- $Q = \{X^c \mid X \subseteq Q_1\}$
- $s = \varnothing$

- $F = \{\varnothing\} \cup \{X \in Q \mid X \sqcap F_1\}$
- $\delta(X, a) = \begin{cases} \{\delta_1(s_1, a)\}^c, & \text{if } X = \varnothing; \\ \hat{\delta}_1(X, a)^c, & \text{otherwise.} \end{cases}$

Note that, since $\varnothing^c = \varnothing$ and the results of each transition are corrected, we have $s \in Q$ and $\delta : Q \times \Sigma \to Q$, as required in the definition of a DFA.

Here is the appropriate notion of trace for this machine, and the lemma characterising δ^*:

Definition 0.26 Suppose $X \in Q$ and $w \in \Sigma^*$. Then a *trace from X on w* is an object of one of the following two forms:

- $(\varnothing \mid w_1, \ldots, w_n)$, where $X = \varnothing$, $n \geq 1$, $w_i \neq \varepsilon$ and $\delta_1^*(s_1, w_i) \in F_1$ for all $i < n$, and $w = \mathsf{concat}\,[w_1, \ldots, w_n]$,
- $(q_1 \mid w_1, \ldots, w_n)$, where $q_1 \in X$, $n \geq 1$, $w_i \neq \varepsilon$ and $\delta_1^*(s_1, w_i) \in F_1$ for all $i < n$, and $w = \mathsf{concat}\,[w_1, \ldots, w_n]$.

$\mathsf{Tr}(X \mid w)$ is the set of all traces from X on w, and $\mathsf{end} : \mathsf{Tr}(X \mid w) \to Q_1$ is the end-point function defined by

$$\begin{aligned} \mathsf{end}(\varnothing \mid w_1, \ldots, w_n) &= \delta_1^*(s_1, w_n) \\ \mathsf{end}(q_1 \mid w) &= \delta_1^*(q_1, w) \\ \mathsf{end}(q_1 \mid w_1, \ldots, w_n) &= \delta_1^*(s_1, w_n) \qquad (n > 1). \end{aligned}$$

Finally, we define

$$E(X, w) = \{\mathsf{end}(t) \mid t \in \mathsf{Tr}(X \mid w)\}.$$

Lemma 0.27 *For all $X \in Q$ and $w \in \Sigma^*$,*

$$\delta^*(X, w) = E(X, w).$$

Proof. Again, we first establish recursion equations for the $E(X, w) \subseteq Q_1$,

$$E(X, \varepsilon) = X \tag{3}$$
$$E(\varnothing, aw) = E(\{\delta_1(s_1, a)\}^c, w) \tag{4}$$
$$E(X, aw) = E(\hat{\delta}_1(X, a)^c, w) \quad (X \neq \varnothing), \tag{5}$$

leaving the reader to check the details. The proof of (3) is given by the table

$t \in \mathsf{Tr}(X \mid \varepsilon)$	$q_1 \in X$
$(q_1 \mid \varepsilon)$	q_1

which I claim defines a total and onto relation (in fact, bijection) between elements $t \in \mathsf{Tr}(X \mid \varepsilon)$ and elements $q_1 \in X$, the connection being $\mathsf{end}(t) = q_1$. The existence of such a relation establishes (3).

The proof of (4) is given by the following table, where each n satisfies $n \geq 2$:

$t \in \mathsf{Tr}(\varnothing \mid aw)$	$t' \in \mathsf{Tr}(\{\delta_1(s_1,a)\}^c \mid w)$	condition
$(\varnothing \mid aw)$	$(\delta_1(s_1,a) \mid w)$	
$(\varnothing \mid aw_1', w_2, \ldots, w_n)$	$(\delta_1(s_1,a) \mid w_1', w_2, \ldots, w_n)$	$w_1' \neq \varepsilon$
$(\varnothing \mid a, w_2, \ldots, w_n)$	$(s_1 \mid w_2, \ldots, w_n)$	$\delta_1(s_1,a) \in F_1$

which I claim defines three parts of a total and onto relation between elements $t \in \mathsf{Tr}(\varnothing \mid aw)$ and elements $t' \in \mathsf{Tr}(\{\delta_1(s_1,a)\}^c \mid w)$ under the indicated conditions, such that $\mathsf{end}(t) = \mathsf{end}(t')$. The existence of such a relation establishes (4).

The proof of (5) is given by the following table, where each n satisfies $n \geq 2$:

$t \in \mathsf{Tr}(X \mid aw)$	$t' \in \mathsf{Tr}(\hat{\delta}_1(X,a)^c \mid w)$	condition
$(q_1 \mid aw)$	$(\delta_1(q_1,a) \mid w)$	
$(q_1 \mid aw_1', w_2, \ldots, w_n)$	$(\delta_1(q_1,a) \mid w_1', w_2, \ldots, w_n)$	$w_1' \neq \varepsilon$
$(q_1 \mid a, w_2, \ldots, w_n)$	$(s_1 \mid w_2, \ldots, w_n)$	$\delta_1(q_1,a) \in F_1$

which I claim defines three parts of a total and onto relation between elements $t \in \mathsf{Tr}(X \mid aw)$ and elements $t' \in \mathsf{Tr}(\hat{\delta}_1(X,a)^c \mid w)$ under the indicated conditions (including $X \neq \varnothing$), such that $\mathsf{end}(t) = \mathsf{end}(t')$. The existence of such a relation establishes (5).

With (3), (4), and (5) established, the proof of the main lemma is now a straightforward induction on w. For $w = \varepsilon$, we have $\delta^*(X,\varepsilon) = X = E(X,\varepsilon)$ by the definitions of δ^* and δ_1^*, and by (3). For $w = aw'$ and $X = \varnothing$, we have

$$\begin{aligned}\delta^*(\varnothing, aw') = \delta^*(\delta(\varnothing,a), w') = \delta^*(\{\delta_1(s_1,a)\}^c, w') \qquad & \text{(defs of } \delta^* \text{ and } \delta\text{)} \\ = E(\{\delta_1(s_1,a)\}^c, w') = E(\varnothing, aw'). \qquad & \text{(IH and (4))}\end{aligned}$$

And for $w = aw'$ and $X \neq \varnothing$, we have

$$\begin{aligned}\delta^*(X, aw') = \delta^*(\delta(X,a), w') = \delta^*(\hat{\delta}_1(X,a)^c, w') \qquad & \text{(defs of } \delta^* \text{ and } \delta\text{)} \\ = E(\hat{\delta}_1(X,a)^c, w') = E(X, aw'), \qquad & \text{(IH and (5))}\end{aligned}$$

completing the proof of the Lemma. ⊣

Theorem 0.28 $L(M_1^*) = L(M_1)^*$.

Proof. Let $w \in \Sigma^*$ be arbitrary. Then

$$\begin{aligned} w \in L(M_1^*) \quad \text{iff} \quad & \delta^*(\varnothing, w) \in \{\varnothing\} \cup \{X \in Q \mid X \sqcap F_1\} \\ & \qquad \text{(def of machine and } L\text{)} \\ \text{iff} \quad & E(\varnothing, w) = \varnothing \vee E(\varnothing, w) \sqcap F_1 \qquad \text{(Lemma)} \\ \text{iff} \quad & w = \varepsilon \vee \exists t \in \mathsf{Tr}(\varnothing \mid w)\ \mathsf{end}(t) \in F_1. \qquad \text{(def of } E\text{)} \end{aligned}$$

Now, if $t \in \mathsf{Tr}(\varnothing \,|\, w)$, then $t = (\varnothing \,|\, w_1, \ldots, w_n)$, where $n \geq 1$, $w_i \neq \varepsilon$ and $\delta_1^*(s_1, w_i) \in F_1$ for all $i < n$, and $w = \mathsf{concat}\ [w_1, \ldots, w_n]$. It follows that $w_i \in L(M_1)$ for all $i < n$, and $\mathsf{end}(t) \in F_1$ means that $w_n \in L(M_1)$ as well, so $w \in L(M_1)^*$ by Theorem 0.12. Conversely, using the same Theorem, if $w \in L(M_1)^*$ then $w = \mathsf{concat}\ l$ for some $l \in [L(M_1)]$. So either $l = [\,]$ and $w = \varepsilon$, or we can assume that $l = [w_1, \ldots, w_n]$, for some $n \geq 1$, where $w_i \in L(M_1)$ and $w_i \neq \varepsilon$ for all i (since any empty string in l can be eliminated without changing the value of the concatenation). Thus, for each i, $\delta_1^*(s_i, w_i) \in F_1$, and so $t = (\varnothing | w_1, \ldots, w_n) \in \mathsf{Tr}(\varnothing | w)$ and $\mathsf{end}(t) \in F_1$, showing, as above, that $w \in L(M_1^*)$ and completing the proof. ⊣

Exercise. Let $\Sigma = \{\mathsf{a}, \mathsf{b}\}$. Draw the machine $M(\mathsf{a})^*$ (Hint: it will have 6 states, 3 of which are final.) What's the smallest machine you can draw that accepts the same language, i.e., $\{\mathsf{a}\}^*$. For the last time, what's going on with these extra states? (For the answer, read on!)

Reachability

If you tried the exercises in the earlier part of this section, you realized that the constructions for union, concatenation, and star tend to produce machines that are much (and often staggeringly) bigger than they have to be. For example, if the machine M_1 has n_1 states and the machine M_2 has n_2 states, then machine $M_1 \cdot M_2$ can have up to $n_1 \cdot 2^{n_2}$ states, meaning that a machine as "simple" as $M(\mathsf{a}) \cdot (M(\mathsf{b}) \cdot M(\mathsf{c}))$, which accepts the singleton language $\{\mathsf{abc}\}$, could have up to $3 \cdot 2^{3 \cdot 2^3}$ or 50,331,648 states, even though it is easy to construct a machine that accepts this language with just 5 states—talk about overkill!

The reason that our constructions are this "wasteful" is that they are completely *general*: since they don't know anything about the constituent machines, they have to include every possibility of interaction between them. In any particular case, however, the constructions will include many states that never get used, in the sense that the machine, staring in its start state and reading an arbitrary string, can never reach those states. For example, the colossal machine mentioned above has only 6 states that can be reached from its start state, and thus 50,331,642 states that might as well not even have been included in the construction!

Let's now formalize the idea of the "reachable" portion of a machine and show that we can simultaneously throw away all of the states that are not reachable and still have a machine that accepts the same language.

Let $M = (Q, s, F, \delta)$ be a DFA. Then we can define a subset $Q_r \subseteq Q$, called the *reachable* states of M, by the following rules:

$$\frac{}{s \in Q_r} \qquad \frac{q \in Q_r \qquad a \in \Sigma}{\delta(q, a) \in Q_r}$$

. Why is Q_r a subset of Q? It follows by a straightforward induction on these

rules, since $s \in Q$ and $\delta(q, a) \in Q$ for every $q \in Q$ and $a \in \Sigma$. Now let M_r be the machine $(Q_r, s_r, F_r, \delta_r)$, where

- $s_r = s$
- $F_r = F \cap Q_r$
- $\delta_r(q, a) = \delta(q, a)$, for $q \in Q_r$ and $a \in \Sigma$.

Clearly, $s_r \in Q_r$ and $F_r \subseteq Q_r$, and the proposition that $q \in Q_r$ and $a \in \Sigma$ imply $\delta_r(q, a) \in Q_r$ is just the second rule above, showing that $\delta_r : Q_r \times \Sigma \to Q_r$ and thus that M_r is a DFA.

Lemma 0.29 *For all $q \in Q_r$ and $w \in \Sigma^*$,*

$$\delta_r^*(q, w) = \delta^*(q, w) \in Q_r$$

PROOF. A straightforward induction on w, which we omit. ⊣

Theorem 0.30 $L(M_r) = L(M)$.

PROOF. Let $w \in L(M_r)$ be arbitrary. Then, by the definitions of the machine and acceptance, $\delta_r^*(s, w) \in F \cap Q_r$, so by the Lemma, $\delta^*(s, w) = \delta_r^*(s, w) \in F$ and thus $w \in L(M)$.

Conversely, let $w \in L(M)$ be arbitrary. The $\delta^*(s, w) \in F$, and so by the Lemma and the definitions, $\delta_r^*(s, w) = \delta^*(s, w) \in F \cap Q_r$, and so $w \in L(M_r)$, as required. ⊣

Exercise. Go back over the exercises in this section and determine which of the states in the machines you constructed were reachable. How does that number compare with the minimal machines you were able to find? Is there still some room for improvement? (This question will be settled completely in Section below.)

DFAs Accept Regular Languages

We established in the previous section (with a brief detour into reachability) that every regular language is accepted by some DFA, so now let's move on in this section to the problem of proving the converse:

Theorem 0.31 *For every finite state machine, M, the language $L(M)$ is regular.*

To do this, I'll first introduce the notion of a *system of proper linear equations* (SPLE) in a finite list of variables and define what it means for a list of languages to satisfy such a system. I'll then show that every SPLE has a unique solution and that every component of that solution is regular. Finally I'll show how we can construct, given a DFA M, a SPLE using the states of M as the variables so that the components of the solution are exactly the $L_q(M)$.

Systems of Proper Linear Equations

A *linear equation* between languages is an equation of the form

$$X = L_1 \cdot X \cup L_2, \tag{6}$$

where L_1 and L_2 are languages and, by convention, the right-hand side is grouped $(L_1 \cdot X) \cup L_2$, as $\cdot$ has higher precedence than $\cup$. A language L is a *solution* to (6) if, naturally, $L = L_1 \cdot L \cup L_2$.

If $\varepsilon \in L_1$, then the solutions to (6) are numerous and not very interesting: for example, $L = \Sigma^*$ is always a solution, and it is just the largest of a generally infinite (even *uncountable*, if that means anything to you) number of solutions. For example, at the extreme, when $L_1 = \{\varepsilon\}$ and $L_2 = \varnothing$ (i.e., when $L_1 = \mathbf{1}$ and $L_2 = \mathbf{0}$), *every* language L is a solution to (6). On the other hand, if we call a language L_1 *proper* when $|w| \geq 1$ for all $w \in L_1$ (which is just a positive way of saying $\varepsilon \notin L_1$, since ε is the only string of length 0), then we have the following remarkable result:

Lemma 0.32 *(Arden's Lemma) For any two languages L_1 and L_2, the language $L = L_1^* \cdot L_2$ is the smallest solution to* (6)*; that is, L is a solution, and if L' is also a solution then $L \subseteq L'$. Moreover, if L_1 is proper, then L is the unique solution.*

Proof. Let L_1 and L_2 be arbitrary languages, and let $L = L_1^* \cdot L_2$. Then we can use the theorems of Section to show that L is a solution to (6):

$$\begin{aligned} L_1 \cdot (L_1^* \cdot L_2) \cup L_2 &= (L_1 \cdot L_1^*) \cdot L_2 \cup L_2 && \text{(by 0.7.3)} \\ &= (L_1 \cdot L_1^*) \cdot L_2 \cup \{\varepsilon\} \cdot L_2 && \text{(by 0.7.2)} \\ &= (L_1 \cdot L_1^* \cup \{\varepsilon\}) \cdot L_2 && \text{(by 0.7.5)} \\ &= L_1^* \cdot L_2. && \text{(by 0.9)} \end{aligned}$$

To show that it's always the smallest solution, suppose $L' = L_1 \cdot L' \cup L_2$, and let's show by induction on $w_1 \in L_1^*$ that, for every $w_2 \in L_2$, we have $w_1 \cdot w_2 \in L'$. If $w_1 = \varepsilon$ and $w_2 \in L_2$, then

$$w_1 \cdot w_2 = w_2 \in L_2 \subseteq L_1 \cdot L' \cup L_2 = L'.$$

And if $w_1 = w \cdot w_1'$, with $w \in L_1$ and $w_1' \in L_1^*$ and $w_2 \in L_2$, then we have, using associativity of $\cdot$ and the IH,

$$w_1 \cdot w_2 = (w \cdot w_1') \cdot w_2 = w \cdot (w_1' \cdot w_2) \in L_1 \cdot L' \subseteq L_1 \cdot L' \cup L_2 = L'.$$

Finally, let's show that if L_1 is proper, then L is also the largest—and thus *only*—solution to (6), in that if L' is also a solution, then $L' \subseteq L$. So, assume L_1 is proper and $L' = L_1 \cdot L' \cup L_2$. By Theorem 0.7, we have

$$L_1 \cdot L = L_1 \cdot (L_1^* \cdot L_2) = (L_1 \cdot L_1^*) \cdot L_2 \subseteq L_1^* \cdot L_2 = L,$$

and thus $L_1 \cdot L \subseteq L$. I will now prove by strong induction on $|w|$ that

$$w \in L' \text{ implies } w \in L. \tag{7}$$

So let n be a natural number, assume (7) holds for all strings w' with $|w'| < n$, let w be such that $|w| = n$, and assume $w \in L'$. Since $L' = L_1 \cdot L' \cup L_2$, either $w \in L_1 \cdot L'$ or $w \in L_2$. In the first case, there exist $w_1 \in L_1$ and $w' \in L'$ such that $w = w_1 \cdot w'$. Since L_1 is proper, we have $|w_1| \geq 1$, and so $|w'| < n$. Therefore, by the IH, $w' \in L$, and so $w = w_1 \cdot w' \in L_1 \cdot L \subseteq L$. In the second case, we use $w = \varepsilon \cdot w$ and $\varepsilon \in L_1^*$ to conclude $w \in L_1^* \cdot L_2 = L$. ⊣

We will say that the equation (6) is *proper* if L_1 is proper. Thus, Arden's Lemma gives us a unique solution to any proper linear equation. Notice, moreover, the crucial point that if L_1 and L_2 are regular, then the unique solution to (6) is also regular, since it is constructed from L_1 and L_2 using Kleene star and concatenation.

Now that we have a way to solve proper linear equations in one variable, we will extend the method to systems of proper linear equations of the form

$$\begin{array}{lcl}
X_1 &=& L_{1,1} \cdot X_1 \cup L_{1,2} \cdot X_2 \cup \cdots \cup L_{1,n} \cdot X_n \cup L'_1 \\
X_2 &=& L_{2,1} \cdot X_1 \cup L_{2,2} \cdot X_2 \cup \cdots \cup L_{2,n} \cdot X_n \cup L'_2 \\
& & \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
X_n &=& L_{n,1} \cdot X_1 \cup L_{n,2} \cdot X_2 \cup \cdots \cup L_{n,n} \cdot X_n \cup L'_n,
\end{array} \tag{8}$$

with n equations in n variables $X_1, X_2, \ldots, X_n$, where every $L_{i,j}$ is proper. Here is the formal definition:

Definition 0.33 A *system of n proper linear equations* (or n-SPLE), where $n \geq 0$, is a pair $(\{L_{i,j}\}_{ij}, \{L'_i\}_i)$, where

- $\{L_{i,j}\}_{ij}$ is an n-by-n matrix of proper languages, called the *coefficients* of the system, and
- $\{L'_i\}_i$ is an n-vector of languages, called the *constants* of the system.

An n-vector of languages, $\{L_i\}_i$, is a *solution* to this system if, for every i,

$$L_i = \big(\bigcup_{j=1}^{n} L_{i,j} \cdot L_j\big) \cup L'_i. \tag{9}$$

Now, Arden's Lemma gives us a recursive method to solve any n-SPLE: if $n = 0$, then the system is empty, and has an empty solution; if $n > 1$, then we

- rewrite the first equation in (8) to the form

  $$X_1 = L_{1,1} \cdot X_1 \cup (L_{1,2} \cdot X_2 \cup \cdots \cup L_{1,n} \cdot X_n \cup L'_1),$$

  and use Arden's Lemma to get a solution for X_1 in terms of the other variables:

  $$X_1 = L_{1,1}^* \cdot (L_{1,2} \cdot X_2 \cup \cdots \cup L_{1,n} \cdot X_n \cup L'_1); \tag{10}$$

- substitute this solution in for the other instances of X_1 in (8), and then use the properties of $\cdot$ and $\cup$ to rearrange the result to get an $(n-1)$-SPLE, $(\{\bar{L}_{i,j}\}_{ij}, \{\bar{L}'_i\}_i)$, where we find that

$$\bar{L}_{i,j} = L_{i,1} \cdot L_{1,1}^* \cdot L_{1,j} \cup L_{i,j} \qquad (i,j \in \{2,\ldots,n\}), \tag{11}$$

$$\bar{L}'_i = L_{i,1} \cdot L_{1,1}^* \cdot L'_1 \cup L'_i \qquad (i \in \{2,\ldots,n\}); \tag{12}$$

- recursively solve this $(n-1)$-SPLE to get solutions $L_2, \ldots, L_n$ for the variables $X_2, \ldots, X_n$; and finally
- extend this solution to include L_1, which we get as the value of X_1 in (10) after substituting $L_2, \ldots, L_n$ in for the variables $X_2, \ldots, X_n$.

The following theorem establishes that this method indeed produces a solution:

Theorem 0.34 *Every n-SPLE $(\{L_{i,j}\}_{ij}, \{L'_i\}_i)$, for $n \geq 0$, has a unique solution, which can be determined using the method above.*

Proof. By induction on n. The case $n = 0$ is trivial, so suppose $n > 0$ and let $\{\bar{L}_{i,j}\}_{ij}$ and $\{\bar{L}'_i\}_i$ be given by (11) and (12). For every $i, j \geq 2$, the language $\bar{L}_{i,j}$ is proper, since $L_{i,1}$ and $L_{i,j}$ are proper, and so $(\{\bar{L}_{i,j}\}_{ij}, \{\bar{L}'_i\}_i)$ is indeed an $(n-1)$-SPLE. By the IH, it has a unique solution $\{L_i\}_i$ such that

$$L_i = \big(\bigcup_{j=2}^{n} \bar{L}_{i,j} \cdot L_j\big) \cup \bar{L}'_i \qquad (i \in \{2,\ldots,n\}). \tag{13}$$

Define

$$L_1 = L_{1,1}^* \cdot \Big((\bigcup_{j=2}^{n} L_{1,j} \cdot L_j) \cup L'_1\Big).$$

We need to show that $\{L_i\}_i$ $(1 \leq i \leq n)$ is a solution to the original system, i.e., satisfies (9) for every $i \geq 1$. The case $i = 1$ follows directly from the definition of L_1 and Arden's Lemma. The cases $i > 1$ follow from the definition of

L_1, the properties of $\cdot$ and $\cup$, and (11), (12), and (13), as follows:

$$
\begin{aligned}
&(\bigcup_{j=1}^{n} L_{i,j} \cdot L_j) \cup L'_i \\
&\qquad = L_{i,1} \cdot L_1 \cup (\bigcup_{j=2}^{n} L_{i,j} \cdot L_j) \cup L'_i \\
&\qquad = L_{i,1} \cdot \Big(L^*_{1,1} \cdot \Big((\bigcup_{j=2}^{n} L_{1,j} \cdot L_j) \cup L'_1\Big)\Big) \cup (\bigcup_{j=2}^{n} L_{i,j} \cdot L_j) \cup L'_i \\
&\qquad = (\bigcup_{j=2}^{n} L_{i,1} \cdot L^*_{1,1} \cdot L_{1,j} \cdot L_j) \cup L_{i,1} \cdot L^*_{1,1} \cdot L'_1 \cup (\bigcup_{j=2}^{n} L_{i,j} \cdot L_j) \cup L'_i \\
&\qquad = (\bigcup_{j=2}^{n} L_{i,1} \cdot L^*_{1,1} \cdot L_{1,j} \cdot L_j) \cup (\bigcup_{j=2}^{n} L_{i,j} \cdot L_j) \cup \bar{L}'_i \\
&\qquad = (\bigcup_{j=2}^{n} L_{i,1} \cdot L^*_{1,1} \cdot L_{1,j} \cdot L_j \cup L_{i,j} \cdot L_j) \cup \bar{L}'_i \\
&\qquad = (\bigcup_{j=2}^{n} (L_{i,1} \cdot L^*_{1,1} \cdot L_{1,j} \cup L_{i,j}) \cdot L_j) \cup \bar{L}'_i \\
&\qquad = (\bigcup_{j=2}^{n} \bar{L}_{i,j} \cdot L_j) \cup \bar{L}'_i, \\
&\qquad = L_i
\end{aligned}
$$

completing the proof. ⊣

From DFAs to SPREs

Variants of DFAs

In this section, we look at several variants of the notion of DFA that seem to add additional power to the model, thereby making it easier to construct machines accepting particular languages, and possibly also increasing the set of languages accepted. In each case, however, we see by means of a reduction that the DFA variant defines the same set of languages as the basic DFAs. To illustrate the benefits of the additional descriptive power of these machines, we revisit the proof in the previous section for each variant, showing how it makes the constructions simpler.

Nondeterministic DFAs

Adding ε-moves

Backward DFAs

More closure properties of regular languages

In this section, we show, as a consequence of the equivalence proved in Sections and , that regular languages, besides being closed under union, concatenation, and Kleene star, are also closed under intersection, set difference, reversal, homomorphism, reverse homorphism, and quotient.

Intersection and Difference

Reversal

Homomorphisms

Quotients

Beyond Regularity

The Pumping Lemma

In this section, we prove a basic result that allows us to show that certain languages are *not* regular.

The Myhill-Nerode Theorem

In this section, we prove a characterization of regular languages that gives us an even more powerful—and theoretically completely general—method for proving that languages are not regular, which moreover has the very useful side-effect of giving us a method to construct the *minimal* DFA accepting a given regular language. The method defines, for any language L, an equivalence relation $\equiv_L$ on Σ^*, and then constructs a "machine" M whose states, Q, are the equivalence classes of $\equiv_L$. The result is that L is regular iff Q is *finite*, in which case M is a DFA, $L(M) = L$, and M the minimum number of states of any machine accepting L.

Context-Free Grammars

Push-Down Automata

Other topics

Extended Grammars

Grammars as we've presented them are completely general, but somewhat tedious to use in the specification of real programming languages. Here we

present a specification for extended grammars, with several features that make them suitable for real-world usages. Note that an extended grammar can always be translated into an equivalent basic grammar; we are not adding any *power* to our grammars, just convenience.

We add the following constructs to grammar definitions:

- Optionality: Instead of writing $\langle A \rangle \mid \varepsilon$ we just write $\langle A \rangle ?$.
- Zero-or-more repetition: Instead of writing a recursive rule $\langle A \rangle ::= \varepsilon \mid \langle A \rangle$ we can just write $\langle A \rangle^*$.
- One-or-more repetition: Instead of writing $\langle A \rangle \langle A \rangle^*$ we write $\langle A \rangle^+$.
- Bounded repetition: The construct $\langle A \rangle \{m, n\}$ indicates that we require *at least* m, and *at most* n, $\langle A \rangle$'s. Similarly, we can write $\langle A \rangle \{m, \}$ to indicate that we require at least m, but with no upper bound, and we can write $\langle A \rangle \{, n\}$ as a shorthand for $\langle A \rangle \{0, n\}$.

Lexical analysis

In our discussions of parsing programming languages, we've glossed over the problem of translating an undistinguished sequence of *characters* into a sequence of *symbols* (or tokens). Lexical analysis is the process of grouping characters into symbols. In most programming languages it is simple enough that it can be accomplished by a variation on DFA-based recognition: we build a DFA that recognizes all the different classes of symbols (e.g., operators, identifiers, numeric literals, string literals, comments, etc.). Each class of symbol results in the machine terminating in a distinct final state (i.e., there is a separate final state for string literals vs. comments, etc.). The machine is built so that it will accept the symbol at the *beginning* of the input string.

Recursive descent parsing

Recursive descent parsing is a technique for implementing LL grammars by hand, by writing a set of mutually-recursive procedures. It has the advantage of being (relatively) easy to construct by hand, and of allowing good error detection, recovery, and reporting.

For LL(k) grammars, we can construct a *predictive* recursive descent parser. A predictive parser is one in which the choice of which nonterminal should be parsed next is entirely and unambiguously determined by some lookahead at the next k symbols, where k is fixed and determined by the grammar. In order to construct a predictive parser, we must compute, for each nonterminal in our grammar, the sequence of lookahead symbols that uniquely identify that nonterminal. k will then be the maximum of the lengths of these sequences.

If k is not fixed for a grammar, we can still make do by using *backtracking*. Backtracking is similar to lookahead, but more generation (and computationally more expensive). With backtracking, whenever we have a choice as to which

nonterminal to parse next, we "mark our place" in the input stream. We then proceed to try to parse the first choice. If it fails, we return to our marked place and then try to parse the next choice. This implies that the same portion of the input stream may be parsed many times; in fact, backtracking can lead to exponential-time parsing. This can sometimes be reduced by memoization.

The question arises as to what to do if *multiple* parses in a choice succeed. If this happens, the grammar is ambiguous, and the ambiguity must be resolved in some way. One simple example of an ambiguous parse is nested `if-else` constructs:

```
if(condition1)
if(condition2)
body1
else
body2
```

Although we are used to the rule that `else`-clauses apply to the *textually-nearest* `if`, grammatically, there is an ambiguity: should this code be parsed as

```
if(condition1)
  if(condition2)
    body1
  else
    body2
```

or as

```
if(condition1)
  if(condition2)
    body1
else
  body2
```

We resolve the ambiguity by applying the extra-grammatical rule given above, choosing the first ("nearest-if") parse.

Parsing expression grammars

One class of grammars in which this kind of "nearest-match" rule is automatically applied are *parsing expression grammars*. PEGs replace the nondeterministic choice operator `|` of traditional grammars with the *ordered choice* operator `/`. Ordered choice means that the *first* choice which succeeds is used, thus, there is no ambiguity due to multiple successful choices, as the first successful choice prevents any later choices from being attempted. Ordered choice, for a suitably-written grammar, automatically encodes the "nearest-match" rule that applies to `if-else` and many other common programming language constructs.

Because the ordered choice operator is sometimes too restrictive, PEG parsers allow two additional constructs:

- $?\langle A \rangle$ implements non-consuming lookahead. $?\langle A \rangle$ succeeds if $\langle A \rangle$ does, but even if it succeeds it consumes no input.
- $!\langle A \rangle$ implements negative non-consuming lookahead. $!\langle A \rangle$ succeeds if $\langle A \rangle$ *fails*, but again, it never consumes any input.

PEG parsers are even easier to write by hand than recursive descent parsers, as they avoid the need to compute the lookahead strings for each nonterminal. However, PEG parsers also suffer from exponential parsing times in the worst case. An extension to PEG parsers is to memoize the results of the parse at each position in the input string. Later parses starting from the same position can then reuse the stored parse results. A parser implemented in this way is a *Packrat* parser; Packrat parsers run in linear-time in the worst case, but naively implemented have dramatically higher storage costs. It's possible to greatly reduce this cost for most grammars by "pruning" the memoized parse results when they are no longer needed. This can be done by manually inserting markers into the grammar, indicating when the grammar has committed to a parse and thus earlier parse results will never be backtracked into. It is also possible to insert these markers automatically, so that an unmodified PEG can be used with Packrat parsing with workable storage costs.

(Packrat parsers also support a number of interesting extensions, for example, support for left-recursion and ...)

Earley parsing

Pratt parsing