PART I

THE FINITE STATE PARADIGM

Chapter 1

Regular Expressions and Acceptors

Section 1.0: Introduction

Throughout this book we will be interested in sequences of symbols, including functions acting on them, and sets of "associated" sequences. The individual symbols themselves are of secondary importance, and we choose to leave them abstract and indefinite for the most part. The symbols may be the binary set {0,1}, and indeed, all of electronic digital computing is eventually reduced to such sequences. At another level, we may be interested in the sequences of ASCII characters that constitute a legal, say, Pascal program. Another option is that each of the possible bit combinations of a 32-bit computer word may be regarded as *one* of our symbols, yielding a large (but finite) collection of primitive symbols. We usually just assume that there is some pre-determined finite, non-empty set Σ that is the universe of distinct atomic symbols — this set is referred to as the **alphabet** and its elements are often called *letters*.

In fact, many details are omitted in the models we consider. Our intent is to adopt a level of abstraction that helps to provide a clear focus on broadly applicable limitations. Abstraction is sometimes regarded as complicating understanding, but quite the contrary is the case. An abstraction simply focuses on one collection of interrelated matters while excluding others. Abstraction is a useful tool, even in everyday matters. One excellent illustration of this is a road map — it omits numerous details (e.g., speed

limits, stop light location, etc.) and thereby allows undivided attention to be directed to the selection of a route. Our use of abstraction is for an analogous purpose — to make some things more clear while deferring consideration of others. As we all know, in computing every detail must be precisely correct. But just this fact, when combined with the enormous number of details involved in many computing applications, implies that we must find abstractions that permit us to avoid the consideration of everything at once.

Eventually any computing task will be accomplished using some specific machine. But one does not initiate problem analysis at the machine level. Before such details can be resolved, we need to organize our thinking and devise the overall approach to be taken to complete the task. In this book we are concerned with computational models at a high level of abstraction. Thus the insights and results we develop are valid for a great variety of specific computing systems. Since we will proceed in a completely machine independent manner, but also wish to retain complete precision in our thinking, we are led to describing the concepts we use with precise definitions, and to following a rigorous process of deduction to understand the properties we can expect to observe in corresponding systems.

To draw an analogy with familiar computational descriptions — when we write a program, we have a finite description (the program) of an infinite entity (the program's "meaning"). The meaning of the program is the collection of all the computations it will carry out when presented with its various valid inputs. When we design a program, we seek to understand this entire collection of computations by examination of its finite description. Overcoming the great gap between a finite static description and a complex dynamic behavior is a dominant issue in computing. Developing means to facilitate this kind of insight and the determination of intrinsic barriers for various models of computing will be our goal throughout this book.

The computing systems we study can be categorized in several ways. One kind, the "transducer", responds to each input sequence by giving an output sequence, and several varieties of these devices are considered in later chapters. This abstraction conforms to our natural intuitive view of the input/output character of computing. For now we pursue a simpler model than that — devices that serve to simply identify a distinguished collection of sequences. Such models may in turn either describe how to "generate" an arbitrary element of the collection, or how to "recognize" whether or not an

arbitrary candidate string belongs to the collection. For recognizers there is a single binary-valued response for each input sequence; these responses are normally referred to as *acceptance* and *rejection*. We will see later that this is not as restrictive a view of computation as might appear at first glance.

Section 1.1: Regular expressions

In this chapter we consider some elementary computing models that simply describe a collection of sequences. This is a somewhat unusual place to begin the technical development. However, the first mechanism we consider names the objects of central consideration in this part of the book, so it is an appropriate starting point. It is a generation-oriented model, and a complementary recognition model is developed in the next section.

For a given set of symbols Σ, we use the notation Σ^* to refer to the set of all finite sequences (sometimes called *strings*) over Σ. Thus if $\Sigma=\{0,1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. Hence Σ^* is an infinite set. We introduce and consistently use the notation ε to denote the null (or empty) sequence which would be invisible without a special notation. Each sequence $x\in\Sigma^*$ has a **length**, written len(x), which is the number of elements in the sequence. Thus $\text{len}(\varepsilon)=0$, $\text{len}(101)=3$, etc. For sequences $x,y\in\Sigma^*$ we write **sequence concatenation** simply as the juxtaposition xy. Of course, $\varepsilon x = x\varepsilon = x$ and $(xy)z = x(yz)$ for all $x,y,z\in\Sigma^*$. Concatenation has the sense of a multiplying operation for strings (with ε as identity), and provides the basis for defining the "power" notations. Thus for $x\in\Sigma^*$, we inductively define:

$x^0 = \varepsilon$, and

$x^{n+1} = (x^n)x$, for $n\geq 0$.

For example, instead of 0000, we usually write 0^4, and for 010101, we may write $(01)^3$. Since concatenation is associative, the power notation satisfies the familiar law of exponents, $x^n x^m = x^{n+m}$ for all $x\in\Sigma^*$ and $n,m\geq 0$.

Any subset L of sequences, $L \subseteq \Sigma^*$, is referred to as a **language**. Languages that may be of interest in various computing circumstances are sets of sequences such as:

- the set of all odd parity sequences in $\{0,1\}^*$,
- the set of all valid Pascal identifiers in {Pascal character set}*, or
- the set of all legal Pascal programs in {Pascal character set}*.

It often takes some time to sink in, but it is critical to recognize the set nature of the definition of "language" — either the omission of *any* intended strings, or the inclusion of *any* prohibited strings would be regarded as providing an utterly incorrect description. We are interested in languages as precisely exact units.

When considering sets, two may be distinguished. One is the *universe* that contains all elements, and every set under consideration is a subset of this set. For languages this is Σ^* for an appropriate alphabet Σ. The other is the *empty set* which has no elements and is a subset of every set under consideration. We use the usual notation, $\varnothing$, for the empty set. Sometimes the empty set and the null sequence are confused with one another, so be careful about this — while they are analogous in the sense of "having no elements", they are different types of entities altogether, one is a set and the other a sequence.

There are three operations on languages that are of fundamental utility. The first of these is the familiar operation of set union. We now define the other two. The second of the operations, called **language concatenation**, combines two languages $L_1, L_2 \subseteq \Sigma^*$ and is defined as $L_1 \bullet L_2 = \{xy \mid x \in L_1$ and $y \in L_2\}$. So this operation elevates concatenation to the level of sets of strings by producing all possible combinations of sequence concatenations for strings chosen from each of the languages. Notice that $\{\varepsilon\} \bullet L = L \bullet \{\varepsilon\} = L$ and $\varnothing \bullet L = L \bullet \varnothing = \varnothing$ for all $L \subseteq \Sigma^*$. Language concatenation has the sense of a multiplying operation for languages (with $\varnothing$ as zero and $\{\varepsilon\}$ as identity), and provides the basis for also defining the "power" notations for languages. For $L \subseteq \Sigma^*$, we inductively define:

$L^0 = \{\varepsilon\}$, and

$L^{n+1} = L^n \bullet L$ for $n \geq 0$.

Thus $L^1 = L$, $L^2 = L \bullet L$, L^3 is L concatenated with itself three times, and consists of all strings which can be formed by the concatenation of any three

strings chosen from L, etc. Language concatenation is also associative, so the power notation for languages satisfies the law of exponents, $L^n \bullet L^m = L^{n+m}$ for all $L \subseteq \Sigma^*$ and n,m≥0.

Example 1.1.1.

With alphabet Σ={0,1}, consider the three languages L_1 = {1, 10}, L_2 = {0,00}, and L_3 = {ε, 10, 1010, 101010, … } = { $(10)^n$ | n≥0}. Then $L_1 \bullet L_2$ = {10, 100, 1000}, $L_2 \bullet L_1$ = {01, 010, 001, 0010}, and $L_2 \bullet L_3$ = {0, 00, 010, 0010, 01010, 001010, … } = {$0(10)^n$ | n≥0} ∪ {$00(10)^n$ | n≥0}. L_1^2 = {11, 110, 101, 1010} = {1^2, $1^2$0, 101, $(10)^2$}.
❑

The third operation, called **Kleene closure**, applies to a language $L \subseteq \Sigma^*$ and is defined as $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ \ldots = \bigcup_{n=0}^{\infty} L^n$. This operation yields the set of all strings that can be formed from an arbitrary number of repetitions of strings selected from L. We will also have occasion to utilize a closely related operation, **positive closure,** that is defined as $L^+ = L \bullet L^* = L^1 \cup L^2 \cup L^3 \cup \ \ldots = \bigcup_{n=1}^{\infty} L^n$, and which differs from L^* by only ε.

Example 1.1.2.

Continuing Example 1.1.1, notice that $\{0, 1\}^* = \{0, 1\}^0 \cup \{0, 1\}^1 \cup \{0, 1\}^2 \cup \ldots = \{\varepsilon\} \cup \{0, 1\} \cup \{00, 01, 10, 11\} \cup \ldots$ = {ε, 0, 1, 00, 01, 10, 11, … } = all sequences. Hence our prior use of the notation Σ^* for all finite sequences is consistent with the definition of Kleene closure. Also, $\{10\}^* = \{10\}^0 \cup \{10\}^1 \cup \{10\}^2 \cup \ldots = \{\varepsilon\} \cup \{10\} \cup \{1010\} \cup \ldots$ = {ε, 10, 1010, … } = L_3 Lastly, $L_1^* = \{1, 10\}^0 \cup \{1, 10\}^1 \cup \{1, 10\}^2 \cup \ldots = \{\varepsilon\} \cup \{1, 10\} \cup \{11, 110, 101, 1010\} \cup \ldots$ = {ε, 1, 10, 11, 110, 101, 1010, … }.
❑

The operations we have introduced are frequently encountered in day-to-day computing. For instance, they are used in many text editors to specify search strings, and they are used (sometimes informally) to describe the

syntax of various programming language constructs (e.g., Pascal identifiers). We will shortly present additional examples.

We are going to be considering ways of describing sets of sequences. For finite sets, an obvious solution is to just write down each of the individual sequences. But for most cases of interest, the sets are *not* finite — for example valid Pascal identifiers, or legal Pascal programs. Since we cannot simply enumerate all the elements of an infinite set, we must seek other means to describe such sets. Generally we will be investigating ways in which we can construct useful finite descriptions of (potentially) infinite objects. A central issue will be what information about an object can be extracted from its description, and how the extraction can be accomplished.

Our first mechanism is referred to as the language of *regular expressions*. This mechanism can be viewed as a meta-language — a language whose purpose is to describe other languages. As a functioning language it has rules of syntax, and a meaning which is associated with each of its valid utterances. First we introduce its rules of syntax.

Definition 1.1.1: Given an alphabet Σ we assume several (seven to be exact) additional (meta) symbols which must not belong to Σ, namely $\{\varepsilon, \varnothing, +, \bullet, ^*, (,)\}$. The **regular expressions over Σ** are finite sequences containing this extended set of symbols, and are determined inductively as follows:

(i) ε, $\varnothing$, and each $\lambda \in \Sigma$ is a regular expression,

(ii) if α and β are regular expressions, then so are $(\alpha+\beta)$, $(\alpha\bullet\beta)$, and (α^*),

(iii) nothing is a regular expression unless it follows from finitely many applications of (i) and (ii).

Definition 1.1.1 provides precise rules for well-formed regular expressions. Actually, it defines them in fully parenthesized form, a restriction we will relax shortly. The meaning to be associated with each valid regular expression has yet to be specified. We are going to regard each such expression as the (finite) description of a (possibly infinite) language over Σ^*. We wish to carefully consider which languages can be described with regular expressions, and what properties such languages must possess. Regular expression are descriptions of languages built-up from letters of the alphabet using concatenation, union, and Kleene closure. Formally we define

their meaning inductively, following the structure provided by Definition 1.1.1, and using the operations on languages that were introduced previously.

Definition 1.1.2: With each regular expression α over the alphabet Σ, we associate a language $\mu(\alpha) \subseteq \Sigma^*$ that is its **meaning** as follows:

$\mu(\varepsilon) = \{\varepsilon\}$,
$\mu(\varnothing) = \varnothing$,
$\mu(\lambda) = \{\lambda\}$ for each $\lambda \in \Sigma$,
$\mu(\alpha+\beta) = \mu(\alpha) \cup \mu(\beta)$,
$\mu(\alpha \bullet \beta) = \mu(\alpha) \bullet \mu(\beta)$,
$\mu(\alpha^*) = (\mu(\alpha))^*$.

Definition 1.1.3: A language $L \subseteq \Sigma^*$ is **regular** if there exists a regular expression α so that $L = \mu(\alpha)$. Also, we say that two regular expressions α and β are **equivalent**, written $\alpha \equiv \beta$, if $\mu(\alpha) = \mu(\beta)$ [and of course, $\equiv$ satisfies the conditions for an equivalence relation on the collection of regular expressions].

Regular expressions were invented by the logician Stephen Kleene (as acknowledged by the name for the '*' operation noted above) in the early 1950s in his study of formal systems called automata. Indeed this was the origination of the investigations we pursue in this chapter. However, since that time regular expressions have come to be widely used for practical purposes such as specifying programming language syntax and describing search patterns in text editors and string processing systems. In the past few years there has been an amazing proliferation of “scripting languages” (e.g., awk, sed, Perl, Tcl, Python, Rexx). These languages include regular expressions as an integral part, and their expressiveness is profoundly effected by those facilities. Regular expression facilities have become so prevalent that the POSIX standard for operating systems [IEE 93] includes a section on regular expressions, and a book entirely devoted to regular expressions in this context has recently been published ([Fri 97]). While there are numerous embellishments in the programming context, this model is in essence the same one that we investigate from a conceptual perspective here.

The objects of interest here are languages (i.e., subsets of Σ^*), and their descriptions are given by regular expressions. Notice that all the finite languages are clearly regular — by concatenating letters, any string can be formed, and by taking the union of individual strings (i.e., singleton sets), any finite set can be formed. Also, certain properties of these languages are readily apparent. For example, inherent in their definition, we see that regular languages are closed under union, language concatenation, and Kleene closure — that is, performing one of these operations on regular sets must produce another regular set. Briefly put, the regular languages constitute the smallest family of languages which contain the finite languages and are closed under these three operations (union, language concatenation, and Kleene closure).

The careful reader may have observed that there is some subtlety in reading the meaning equations of Definition 1.1.2. For example, in the equation $\mu(\varnothing) = \varnothing$, the occurrence of the symbol $\varnothing$ on the left-hand side is one of the meta-symbols of the language of regular expressions, whereas the occurrence of $\varnothing$ on the right-hand side is the usual notation denoting the empty set. Similar observations can be made about other equations. This is perhaps also a good time to raise a caution about the "overloading" of the symbol '='. For instance, when we write $\mu(\alpha) = \mu(\beta)$ we are referring to an equality between sets of strings, and one must understand set equality. However, when we write $\alpha = \beta$ for regular expressions α and β, we are referring to sequences of symbols, and we are expressing sequence equality.

Finally, in order to provide more convenient and usable notation, we relax our definition of regular expressions just a bit. Writing regular expressions in fully parenthesized form as they are formally defined interferes with readability and is tedious. So, we adopt a convention similar to that for expressions in programming languages, and assign each of the three regular expression operations a **precedence**. In the absence of parentheses, Kleene closure has highest precedence, language concatenation has intermediate precedence, and union (+) has lowest precedence. Repeated occurrences of the same operation may be understood in left-to-right order, but since the operations are associative this is not significant. Also, we often omit the operation symbol for language concatenation and simply juxtapose the operands. Notice that with these conventions, we may only be able to

distinguish a regular expression from an ordinary string by the context of its use.

Example 1.1.3: regular expressions using alphabet $\Sigma=\{0,1\}$:

(a) The regular expression $((1\bullet0)\bullet1)$ may be written without parentheses as $1\bullet0\bullet1$ and then simply as 101. Its meaning is $\mu((1\bullet0)\bullet1) = \mu(1\bullet0)\bullet\mu(1) = (\mu(1)\bullet\mu(0))\bullet\mu(1) = (\{1\}\bullet\{0\})\bullet\{1\} = \{101\}$.

(b) The regular expression (0^*) will be written 0^*. Its meaning is $\mu(0^*) = \mu(0)^* = \{0\}^* = \{\varepsilon, 0, 00, 000, \ldots\}$; we may indicate this by $\mu(0^*) = \{0^n \mid n\geq0\}$.

(c) The regular expression $((1\bullet(0^*))\bullet1)$ is written 10^*1. Its meaning is $\mu(10^*1) = (\mu(1)\bullet\mu(0)^*)\bullet\mu(1) = \{1\}\bullet\{0^n \mid n\geq0\}\bullet\{1\} = \{10^n1 \mid n\geq0\}$.

(d) The regular expression $(((1\bullet0)\bullet1)+((1\bullet(0^*))\bullet1))$ is written $101+10^*1$. Then $\mu(101+10^*1) = \mu(101) \cup \mu(10^*1) = \{101\} \cup \{10^n1 \mid n\geq0\} = \{10^n1 \mid n\geq0\}$.

To help provide some intuition about regular expressions, compare the following examples with their corresponding informal English descriptions (taking $\Sigma=\{0,1\}$).

(e) all sequences — $(0+1)^*$

(f) all sequences ending with '1' — $(0+1)^*1$

(g) the three sequences 00, 11, and 101 — $00+11+101$

(h) all sequences with an even number of '1's — $0^*(10^*10^*)^*$.

(i) all sequences where no '0' follows a '1' — 0^*1^*.

(j) all sequences either starting with '0' or ending with '1' — $0(0+1)^* + (0+1)^*1$

(k) all sequences with the same first and last character — $0+1+ 0(0+1)^*0 + 1(0+1)^*1$

(l) all sequences, except for the three sequences 00, 11, and 101 — $\varepsilon+0+1+10+ (01 + 100 + (00+11+101)(0+1))\ (0+1)^*$.

❑

There are many times that a clear and unambiguous English description cannot be given. The purpose of a formalism such as regular expressions is

to provide a description that is precise and unmistakable in all cases. Furthermore, as we will see in the following sections, regular expressions provide the basis for effective automation. The capacity to facilitate algorithmic processes and their analysis guides the nature of the formalisms we seek to develop.

The Kleene closure operation plays a central role here since regular expressions start with finite sets (singleton sets of individual letters actually), and only Kleene closure produces an infinite result from finite arguments. Since a principal goal is to provide finite descriptions for infinite sets, a good understanding of Kleene closure is critical.

There are a number of useful regular expression identities (actually we present schemes for describing identities for many specific regular expressions). Verifying such identities provides a first encounter with the issue of extracting information from descriptions.

Theorem 1.1.1: For any regular expressions α, β and γ,

(i) $\alpha+\beta \equiv \beta+\alpha$,
(ii) $(\alpha+\beta)+\gamma \equiv \alpha+(\beta+\gamma)$,
(iii) $\emptyset+\alpha \equiv \alpha+\emptyset \equiv \alpha$,
(iv) $(\alpha\beta)\gamma \equiv \alpha(\beta\gamma)$,
(v) $\alpha(\beta+\gamma) \equiv \alpha\beta+\alpha\gamma$,
(vi) $(\alpha+\beta)\gamma \equiv \alpha\gamma+\beta\gamma$,
(vii) $\varepsilon\,\alpha \equiv \alpha\,\varepsilon \equiv \alpha$,
(viii) $\emptyset\,\alpha \equiv \alpha\,\emptyset \equiv \emptyset$,
(ix) $\emptyset^* \equiv \varepsilon$,
(x) $(\alpha+\varepsilon)^* \equiv \alpha^*$,
(xi) $\alpha(\beta\,\alpha)^* \equiv (\alpha\,\beta)^*\alpha$,
(xii) $(\alpha^*)^* \equiv \alpha^*$,
(xiii) $(\alpha^*\,\beta^*)^* \equiv (\alpha+\beta)^*$,
(xiv) $(\alpha\,\beta^*)^* \equiv \varepsilon+\alpha(\alpha+\beta)^*$.

The proofs of several parts of Theorem 1.1.1 are immediate from the definitions and the properties of the operations on sets of strings — we omit these, or leave them to exercises. However, as a general matter, proving the equivalence of regular expressions can sometimes require careful attention.

As an example of the care necessary, we present the detailed proof of one of these claims.

Proof of Theorem 1.1.1, part xiii.

We need to establish the set equality $\mu((\alpha^* \beta^*)^*) = \mu((\alpha+\beta)^*)$. This requires that we establish the subset relation in both directions.

Step 1: prove that $\mu((\alpha^* \beta^*)^*) \subseteq \mu((\alpha+\beta)^*)$.

Proof: Suppose that $x \in \mu((\alpha^* \beta^*)^*)$. Then for some $k \geq 0$, $x = x_1 x_2 \ldots x_k$ where $x_i \in \mu(\alpha^* \beta^*)$, $0 \leq i \leq k$. Therefore $x_i = y_i z_i$ where $y_i \in \mu(\alpha^*)$ and $z_i \in \mu(\beta^*)$. But then $y_i = y_{i,1} y_{i,2} \ldots y_{i,m_i}$, and $z_i = z_{i,1} z_{i,2} \ldots z_{i,n_i}$ where $y_{i,j} \in \mu(\alpha)$ and $z_{i,j} \in \mu(\beta)$. Hence $x = y_{1,1} y_{1,2} \ldots y_{1,m_1} z_{1,1} z_{1,2} \ldots z_{1,n_1} y_{2,1} y_{2,2} \ldots y_{2,m_2} z_{2,1} z_{2,2} \ldots z_{2,n_2} \ldots y_{k,1} y_{k,2} \ldots y_{k,m_k} z_{k,1} z_{k,2} \ldots z_{k,n_k}$. But since $y_{i,j} \in \mu(\alpha+\beta)$ and $z_{p,q} \in \mu(\alpha+\beta)$ for all appropriate i,j,p,q, this means $x \in \mu((\alpha+\beta)^*)$.

Step 2: prove that $\mu((\alpha+\beta)^*) \subseteq \mu((\alpha^* \beta^*)^*)$.

Proof: Suppose that $x \in \mu((\alpha+\beta)^*)$. Then for some $k \geq 0$, $x = x_1 x_2 \ldots x_k$ where $x_i \in \mu(\alpha+\beta)$, $1 \leq i \leq k$. Hence either $x_i \in \mu(\alpha)$ or $x_i \in \mu(\beta)$. But if $x_i \in \mu(\alpha)$, then $x_i = x_i \varepsilon \in \mu(\alpha^* \beta^*)$, and if $x_i \in \mu(\beta)$, then $x_i = \varepsilon x_i \in \mu(\alpha^* \beta^*)$. Thus $x_i \in \mu(\alpha^* \beta^*)$ in either case, and so $x \in \mu((\alpha^* \beta^*)^*)$.

❑

As these equivalencies illustrate, a regular expression associates a certain "pattern" with the strings it describes, and this does not exclude the possibility of attaching alternative patterns with exactly the same strings. Properties that may be evident when regarding this set of strings one way may become less obvious with an alternative pattern.

The equivalencies of Theorem 1.1.1 permit us to avoid burdensome string level analysis and make some other deductions relatively easily. For instance, $(\alpha^*+\beta)^* \equiv ((\alpha^*)^* \beta^*)^* \equiv (\alpha^* \beta^*)^* \equiv (\alpha+\beta)^*$ by first applying (xiii), then (xii), and then (xiii) again. It would be desirable to determine a list of regular expression equivalencies that we could use as "axioms" to

prove any other equivalence of interest. Regrettably no finite collection of equivalencies exists that is adequate for this purpose, although by adding some inequalities a finite axiom set can be obtained. It is beyond the scope of our treatment to prove these claims — the interested reader can consult [Sal 69] and [Koz 94]. We will develop an indirect approach to verify (or refute) general identities later.

We have seen that the regular expressions provide a means to describe a variety of languages. We now introduce two other operations on sets of strings that are helpful in understanding the regular languages as well as various other language families to be introduced later.

Definition 1.1.4: Given two alphabets Σ and Δ, a **homomorphism** is a function $h: \Sigma \rightarrow \Delta^*$, that is, from letters of the first alphabet to strings over the second. We immediately extend h to operate as a function on Σ^* by element-wise application — inductively, $h(\varepsilon) = \varepsilon$, and for all $x=\lambda y$ with $\lambda \in \Sigma$ and $y \in \Sigma^*$, $h(x) = h(\lambda)h(y)$; finally we further extend h to operate as a function on languages, where for $L \subseteq \Sigma^*$, $h(L) = \{h(x) \mid x \in L\}$.

Notice that a homomorphism has a finite description — for each of the finitely many letters $\lambda \in \Sigma$, we need only identify the string $h(\lambda) \in \Delta^*$ and this completely determines the behavior of h on all strings, and sets of strings. From its definition, the extension of a homomorphism $h: \Sigma^* \rightarrow \Delta^*$ has the property that $h(\lambda_1\lambda_2 \ldots \lambda_k) = h(\lambda_1)h(\lambda_2) \ldots h(\lambda_k)$ where $\lambda_i \in \Sigma$ $(1 \leq i \leq k)$, and so $h(xy) = h(x)h(y)$ for all $x,y \in \Sigma^*$. Simply stated, a homomorphism allows the renaming of letters as strings.

Example 1.1.4:

Consider $\Sigma = \{0, 1\}$ and $\Delta = \{a, b\}$ with homomorphism $h_1: \Sigma \rightarrow \Delta^*$ defined by $h_1(0) = aba$, $h_1(1) = bb$. Then $h_1(101) = bbababb$, $h_1(001) = abaababb$, and $h_1((0^*1)^*) = ((aba)^*bb)^*$. For another homomorphism consider $h_2: \Sigma \rightarrow \Delta^*$ defined by $h_2(0) = aa$, $h_2(1) = \varepsilon$. Then $h_2(101) = aa$, $h_2(001) = aaaa$, and $h_2((0^*1)^*) = ((aa)^*)^* = (aa)^*$.

❑

In the preceding example we have begun a convenient (and common) abuse of the notation. Instead of writing $L=\mu(\alpha)$ for a language L and regular expression α, we just write $L=\alpha$. Thus we use regular expressions to denote both themselves and the language that is their meaning and rely on the reader using context to determine which we intend.

A homomorphism can serve as a precise language description that is derived from another known language. There is another commonly used mapping mechanism that generalizes homomorphisms by allowing for a choice among the strings that replace each letter.

Definition 1.1.5: Given two alphabets Σ and Δ, a **substitution** is a function $\sigma: \Sigma \rightarrow \wp(\Delta^*)$ [for set X, the notation $\wp(X)$ denotes the collection of all subsets of X, the *power set* of X]. Thus each letter is associated with a (possibly infinite) set of strings, or language. We immediately extend σ to operate as a function on Σ^* by element-wise application — inductively, $\sigma(\varepsilon) = \{\varepsilon\}$, and for all $x=\lambda y$ with $\lambda \in \Sigma$ and $y \in \Sigma^*$, $\sigma(x) = \sigma(\lambda) \bullet \sigma(y)$; then we further extend σ to operate as a function on languages, where for $L \subseteq \Sigma^*$, if $L = \{x_1, x_2, x_3, \ldots\}$, then $\sigma(L) = \sigma(x_1) \cup \sigma(x_2) \cup \sigma(x_3) \cup \ldots$ A substitution σ is called **regular** if for each $\lambda \in \Sigma$, $\sigma(\lambda)$ is regular.

So with a substitution σ, for each letter $\lambda \in \Sigma$ there is a collection of choices of the strings, $\sigma(\lambda) \subseteq \Delta^*$, that may replace that letter. The substitution is described by providing this (finite) association of letters with languages (but the languages need not be finite). To apply a substitution to a string, choices for the individual letters of the string are concatenated together to obtain choices to replace the string; and when the substitution is applied to a set of strings, the choices constituting the set of results are obtained by forming the union of the choices for each of the strings in the set. Since a substitution associates each letter with a language and the language may be infinite, in general substitutions need not always have finite descriptions. However, for a regular substitution each language has a finite description, so we do have finite descriptions of regular substitutions.

Example 1.1.5:

Let $\Sigma = \{0, 1\}$ and $\Delta = \{a, b, c, d\}$ with substitution $\sigma_1: \Sigma \to \wp(\Delta^*)$ defined by $\sigma_1(0) = \mu(aba)$, $\sigma_1(1) = \mu(cd^*)$, where we use regular expressions to describe the set of choices for each letter. Then $\sigma_1(101) = \mu(cd^*abacd^*)$, $\sigma_1(001) = \mu(abaabacd^*)$, and $\sigma_1((0^*1)^*) = \mu(((aba)^*cd^*)^*)$. For another substitution consider $\sigma_2: \Sigma \to \wp(\Delta^*)$ defined by $\sigma_2(0) = \mu((aa)^*)$, $\sigma_2(1) = \mu(\varepsilon+c)$. Then $\sigma_2(101) = \mu((aa)^*+(aa)^*c+c(aa)^*+c(aa)^*c)$, $\sigma_2(001) = \mu((aa)^*(aa)^*(\varepsilon+c)) = \mu((aa)^* (\varepsilon+c))$, and $\sigma_2((0^*1)^*) = \mu((((aa)^*)^* (\varepsilon+c))^*) = \mu(((aa)^* (\varepsilon+c))^*)$. As a last example consider $\sigma_3: \Sigma \to \wp(\Delta^*)$ defined by $\sigma_3(0) = \mu(\varnothing)$, $\sigma_3(1) = \mu(a+b)$. Then $\sigma_3(101) = \sigma_3(001) = \varnothing$, and $\sigma_3((0^*1)^*) = \mu(\varnothing^*(a+b)^*) = \mu((a+b)^*)$.
❑

Notice that for a homomorphism, if we view the string associated with each letter as a singleton set, we can regard a homomorphism as a "deterministic" substitution where there is exactly one choice of string for each letter. In that sense, a homomorphism is just a restricted substitution, and any result that is true for substitutions also holds for homomorphisms.

Lemma 1.1.2: For each $X,Y \subseteq \Sigma^*$ and each substitution (or homomorphism) $\sigma: \Sigma \to p(\Delta^*)$,

(i) $\sigma(X \cup Y) = \sigma(X) \cup \sigma(Y)$,

(ii) $\sigma(X \bullet Y) = \sigma(X) \bullet \sigma(Y)$, and

(iii) $\sigma(X^*) = (\sigma(X))^*$.

The proof is left as an exercise.

As the preceding examples suggest, we have

Theorem 1.1.3: For each regular language $R \subseteq \Sigma^*$ and each regular substitution σ, $\sigma(R)$ is also regular.

Proof of Theorem 1.1.3.

This proof uses what is sometimes called "structural induction" — since R is regular, there is some regular expression, say α, with $R = \mu(\alpha)$. We use

the structure of α to guide the analysis. In particular, we consider the number of operations occurring in α.

Anchor (or basis) step — α involves no operations.

In this case α must be either ε, $\varnothing$, or $\lambda \in \Sigma$. But $\sigma(\{\varepsilon\}) = \{\varepsilon\}$, $\sigma(\varnothing) = \varnothing$, and $\sigma(\lambda)$ is assumed to be regular, so $\sigma(R)$ is regular in each of these cases.

Induction step — assume that $\sigma(\mu(\alpha))$ is regular for each α involving n or fewer operations, and consider $R = \mu(\beta)$, where β involves n+1 operations. Then we must have one of the following three cases:

case (i): $\beta = \gamma + \delta$.

Then γ and δ each involve n or fewer operations, and so by the induction hypothesis, $\sigma(\mu(\gamma))$ and $\sigma(\mu(\delta))$ are regular, and since $\sigma(\mu(\beta)) = \sigma(\mu(\gamma) \cup \mu(\delta)) = \sigma(\mu(\gamma)) \cup \sigma(\mu(\delta))$ by the definition of μ and Lemma 1.1.2, $\sigma(\mu(\beta))$ is regular.

case (ii): $\beta = \gamma \bullet \delta$.

Then γ and δ each involve n or fewer operations, and so by the induction hypothesis, $\sigma(\mu(\gamma))$ and $\sigma(\mu(\delta))$ are regular, and since $\sigma(\mu(\beta)) = \sigma(\mu(\gamma) \bullet \mu(\delta)) = \sigma(\mu(\gamma)) \bullet \sigma(\mu(\delta))$ by the definition of μ and Lemma 1.1.2, $\sigma(\mu(\beta))$ is regular.

case (iii): $\beta = \gamma^*$.

Then γ involves n or fewer operations, and so by the induction hypothesis, $\sigma(\mu(\gamma))$ is regular, and since $\sigma(\mu(\beta)) = \sigma(\mu(\gamma)^*) = (\sigma(\mu(\gamma)))^*$ by the definition of μ and Lemma 1.1.2, $\sigma(\mu(\beta))$ is regular.

Hence no matter which case pertains, the induction is extended and so the result is proven.

❑

Corollary 1.1.4: For each regular language R and each homomorphism h, h(R) is also regular.

Proof of Corollary 1.1.4:

For a homomorphism h, for each $\lambda \in \Sigma$, $h(\lambda)$ is a single sequence from Δ^* and hence may be regarded as a singleton set and thus is regular. Since h is therefore a regular substitution, by Theorem 1.1.3, h(R) is regular.

❑

The proof of Theorem 1.1.3 is based on the inductive definition of regular expressions. It is natural to mimic that inductive definition in the

induction of our proof. The structure of associated regular expressions is used to organize the analysis. This is an approach that we will frequently find useful. Notice that the proof of Theorem 1.1.3 reveals an algorithm for constructing a regular expression for $\sigma(R)$ from the regular expressions for R and $\sigma(\lambda)$, $\lambda \in \Sigma$. Namely, replace each letter λ occurring in the regular expression for R with the corresponding regular expression for $\sigma(\lambda)$.

We now have several operations which when applied to languages known to be regular, are bound to yield another regular language. This can ease the effort needed to show that a language is regular by building on prior knowledge rather than always being forced to start from scratch.

Example 1.1.6.

Pascal identifiers are required to begin with a letter which then can be followed by any number of letters and digits. This is immediately seen to be a regular language since <Pascal identifier> = $\sigma(01^*)$, where σ is the substitution defined by $\sigma(0)$ = <letter> and $\sigma(1)$ = <letter> $\cup$ <digit>, and both <letter> and <digit> are finite and hence regular. Therefore by Theorem 1.1.3, the set of all legal Pascal identifiers is a regular language. Substitutions formally support this abstract approach of thinking of a single token in place of a whole collection of strings that can actually appear.
❑

This section has presented the first "formal model". This is a formal means for describing collections of sequences, and it is not yet apparent how this model should be viewed as describing "computations". We will see that regular expressions can be regarded as the syntax that provides a high level description of certain computational results associated with their meanings. Facilities sufficient to accomplish corresponding computations will be elaborated in the next two sections. At this point we have provided only the first of several steps that are required to complete the picture.

Section 1.2: Finite state acceptors

While some of the properties of the regular languages have begun to emerge, other things are not so clear at this point. For instance, is the complement of every regular language regular? An even more basic question is: given a

string $x \in \Sigma^*$ and a regular expression α over Σ, what is the procedure for determining whether or not $x \in \mu(\alpha)$? For better methods to address such questions, we now proceed to other means of description of sets of strings. At this point it will no doubt appear that we are jumping from one formalism in the previous section, to an entirely different one in this section. From one perspective that is indeed the case, but it will be revealed in the following section that a deep underlying similarity exists.

Given any description of a language, one of the basic questions we naturally expect to be able to answer is whether a given string belongs to the language or not. This is called the *membership question* and a language description is of little practical value if it does not permit us to answer this question. Regular expressions address how to *generate* acceptable sequences, and there is no obvious procedure to follow to answer membership questions — e.g., does 101 belong to $\mu(\varepsilon+0+1+10 + (01 + 100 + (00+11+101)(0+1))\ (0+1)^*)$? The models developed in this section are especially suited to answering membership questions, and turn out to be useful in many other ways as well.

The finite state acceptor model we develop now can be thought of as an abstraction of a simple computing device that has only a fixed internal store (or memory), and responds to input with a single binary output — accept or reject. The input consists of a sequence of symbols that is read and processed one symbol at a time. We wish to describe behavior where the reaction to an individual input symbol may depend on prior input as well as the current symbol itself, so the behavior of the model incorporates internal "memory". We adopt a high level of abstraction in the model so that the analysis is not sensitive to the details of any particular computing system. The internal storage is understood to be an amalgamation of some finite (but unrestricted) number of "configurations". At this first stage, no other resource is available in the model. Such a model probably appears to be quite limited, and indeed there are serious limitations. The finite state models we develop in this section will recur repeatedly in later more complicated models as a basic sub mechanism. So while the limitations will disappear, this model will persist. Also, we will find in Chapter 3 that having only a single binary response to input sequences is not nearly as limiting as it may initially seem.

Definition 1.2.1: A **deterministic transition system** (DTS) is a triple, $T = (S, \Sigma, \delta)$, where S, the **states**, and Σ, the **input alphabet**, are finite non-empty sets, and δ is a function, $\delta: S \times \Sigma \rightarrow S$, the **next-state function**.

A transition system captures the way the configurations of the internal store of a computing device change when a single input symbol is presented to it. The internal memory (the states) permit the model to retain information about past input, and this may effect the reaction to later input symbols. To capture this, we need to describe how the transitions cascade for a *sequence* of inputs.

Definition 1.2.2: If $T = (S, \Sigma, \delta)$ is a deterministic transition system, then the **transition function**, $\delta^*: S \times \Sigma^* \rightarrow S$, is defined (inductively) for all $s \in S$ by: $\delta^*(s, \varepsilon) = s$, and for all $x \in \Sigma^*$ and $\lambda \in \Sigma$, $\delta^*(s, x\lambda) = \delta(\delta^*(s, x), \lambda)$.

Thus if there is no input, the state does not change, and the state transitions for non-null sequences of input symbols are simply determined by the sequential, letter-by-letter, application of the next-state function. The common notational convention is to simply write δ when either δ or δ^* is really intended — the reader must be alert to the context to determine which is meant (in programming language parlance, we are overloading the function symbol δ with both meanings). Since for the common domain of the two functions, $\delta^*(s, \lambda) = \delta(s, \lambda)$, the possibility of confusion with this convention is negligible.

Given the state and input sets, to completely describe a transition system, we need only describe the next-state function. Since the domain of this function is finite, the value of the function for each argument pair can be given in a finite list. Once this is specified, the transitions for all input sequences of arbitrary length are determined.

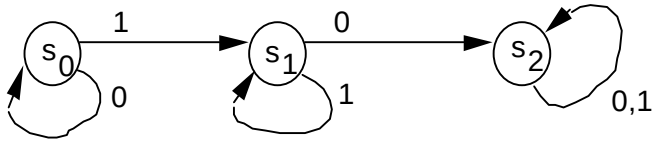
Example 1.2.1.

Take $S = \{s_0, s_1, s_2\}$, and $\Sigma = \{0, 1\}$. We define the next-state function for this transition system by a table that lists the result for each combination of state and symbol in tabular form below.

δ	s_0	s_1	s_2
0	s_0	s_2	s_2
1	s_1	s_1	s_2

This table records the (six) individual state changes: $\delta(s_0, 0) = s_0$, $\delta(s_0, 1) = s_1$, $\delta(s_1, 0) = s_2$, $\delta(s_1, 1) = s_1$, $\delta(s_2, 0) = s_2$, $\delta(s_2, 1) = s_2$.

Our preferred way of presenting transition system examples is by means of *state diagrams* — the states are represented as nodes in a directed graph, the edges are determined by δ and labeled with the input associated with that state change. For the transition system here we have the state diagram



❑

State diagrams are helpful presentations of transition systems. The crucial thing for transition systems is how the states change, and this can be seen clearly from the state diagram (at least for moderate sized examples). It should be noted that the *edges* in the state diagram record the next-state function (δ). The transition function (δ^*) is described by the *paths* (i.e., sequences of edges) of the diagram. The analysis of transition systems generally revolves around determining (global) path properties from the (local) edge information.

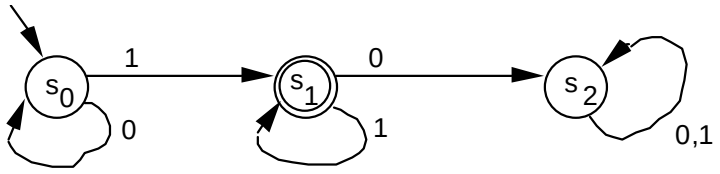
Definition 1.2.3: A **deterministic finite acceptor** (DFA) A is a 5-tuple $A = (S, \Sigma, \delta, s_0, R)$, where (S, Σ, δ) is a deterministic transition system, $s_0 \in S$ is the **start state**, and $R \subseteq S$ is the set of **recognizing** (or **accepting**) states. The **language recognized** (or **accepted**) by A is $L(A) = \{x \in \Sigma^* \mid \delta(s_0, x) \in R\}$. A language $L \subseteq \Sigma^*$ is **DFA-recognizable** if there exists a DFA A so that $L = L(A)$.

Note that since all the sets involved in a DFA are finite, a DFA A provides a finite description of its language, L(A). The cases of primary interest are when L(A) is *not* finite — then we have a finite description of an infinite object and we must base our understanding of this object entirely on this description.

Example 1.2.2.

Consider the transition system of Example 1.2.1. If we additionally identify s_0 as the start state, and $R=\{s_1\}$ as the set of accepting states, then we have an acceptor, say A, that recognizes the language $L(A) = \mu(0^*11^*)$. To verify that A accepts exactly 0^*11^*, we have to construct a set equality argument. We need to observe that every string 0^p1^q ($p\geq 0$, $q\geq 1$) in 0^*11^* is recognized by A — $\delta(s_0, 0^p1^q) = \delta(s_0, 1^q) = \delta(s_1, 1^{q-1}) = s_1$. Also we need to observe that every string not in 0^*11^* is rejected by A — for a string not to be in 0^*11^*, either it has no '1's (i.e., 0^p, $p\geq 0$), or it has a '0' following a '1' (i.e., 0^p1^q0x, where $p\geq 0$, $q\geq 1$, and $x\in(0+1)^*$); but $\delta(s_0, 0^p) = s_0$, and $\delta(s_0, 0^p1^q0x) = \delta(s_1, 1^{q-1}0x) = \delta(s_1, 0x) = \delta(s_2, x) = s_2$ so both kinds of strings are rejected.

We augment state diagrams for transition systems to denote the start state by an unattached edge pointing to it, and we denote the accepting states by double circle nodes. Thus A is depicted by the state diagram



❑

The DFA model provides an abstraction for computing systems where the input is processed letter by letter. The input source could be a keyboard and each keystroke produces one of the letters, or the input source might be a communication line where the characters arrive sequentially one by one, or perhaps a “tape” device of some type which has a unidirectional (forward) movement. After the last letter is input, a console light (or some comparable

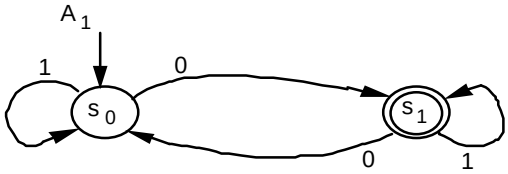
binary signal) will either be on or off to show acceptance or rejection. As a consequence of the abstraction, the DFA analysis techniques will apply to a wide variety of realizations. This model does not yet conform to realistic computing systems, but it will be a useful beginning.

Clearly the DFA-recognizable languages admit a straight-forward procedure for answering the membership question. Given an arbitrary candidate string, all we need to do is simulate the computation of the DFA on the string, and see whether the ending state is accepting or not. This can be done with a table driven algorithm that embeds the state transition table and accepting state list, records current state, and does letter by letter table look-up. Clearly such an algorithm requires only a fixed amount of storage (not varying with the length of the input), and time that is directly proportional to (i.e., linear in) the input length.

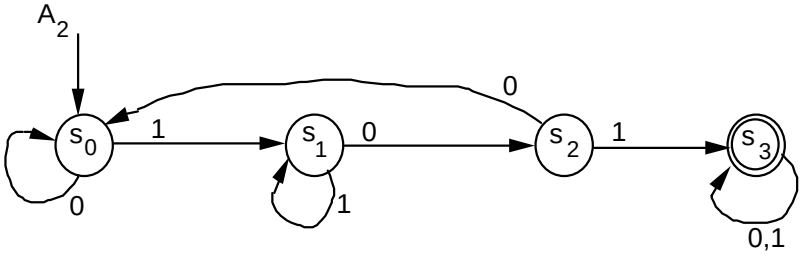
Example 1.2.3

In the three DFAs provided below, the reader should verify the claims about the strings that are recognized (and rejected).

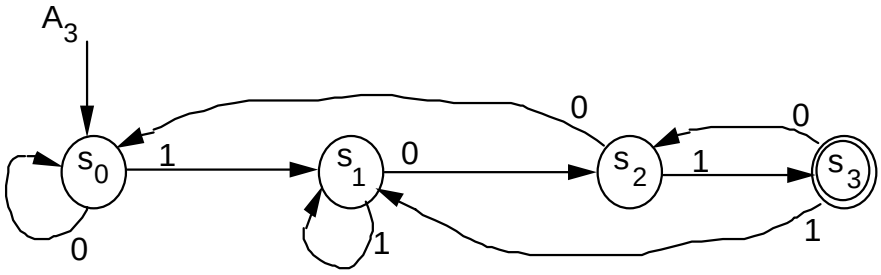
$L(A_1)$ = *all* sequences with an odd number of '0's



$L(A_2)$ = *all* sequences containing subsequence '101'



$L(A_3)$ = *all* sequences ending with subsequence '101'



❑

As is already apparent, and as we will repeatedly observe as we go along, the cycle structure of an acceptor is a key factor in its descriptive power. On the other hand, the cycle structure is also the critical complication in understanding these devices. In fact, this is an inevitable characteristic of all methods of describing computations, (automata or programming languages) — the features that most enhance descriptive power are invariably associated with greater conceptual complication.

Theorem 1.2.1: the complement of each DFA-recognizable language is also DFA-recognizable.

Proof:

This proof is a simple construction. Given L = L(A), where A = (S, Σ, δ, s_0, R) is a DFA, we define DFA B = (S, Σ, δ, s_0, S–R). Then for any $x \in \Sigma^*$, if $\delta(s_0, x) \in R$, then $\delta(s_0, x) \notin S-R$, and conversely. Hence clearly $L(B) = \neg L(A) = \Sigma^* - L(A)$.

❑

Even though Theorem 1.2.1 is very simple technically, it provides a useful way of approaching languages that are described negatively. Providing a description of what is not a string of interest requires that a deduction be made to positively identify strings. Theorem 1.2.1 shows that this is a trivial transformation for DFAs. It is much less clear for regular expressions. We will later see mechanisms for which negative descriptions change the character of the language drastically.

In the next section we will show that the DFA-recognizable languages are synonymous with the regular languages. To establish this equivalence, we need to develop some additional means of analysis. The models we develop next not only provide these additions, but also introduce challenging concepts crucial to more complex later models in this simpler setting.

Definition 1.2.4: A **non-deterministic transition system** (NTS) is a triple, $T = (S, \Sigma, \delta)$, where S, the **states**, and Σ, the **input alphabet**, are finite non-empty sets, and δ is a function, $\delta: S\times\Sigma \rightarrow \wp(S)$, the **next-state function** [for set X, the notation $\wp(X)$ denotes the collection of all subsets of X, the *power set* of X].

A non-deterministic transition system does not uniquely prescribe configuration changes. Instead a collection of possible outcomes is given with the understanding that at any given step of a computation one of these state changes will occur, but the particular possibility that transpires at any moment varies in an undetermined manner. The internal memory (the states) still permit the model to retain information about past input that may effect the reaction to later input symbols. To capture this, we again need to describe how the uncertain transitions cascade for a sequence of inputs.

Definition 1.2.5: If $T = (S, \Sigma, \delta)$ is a non-deterministic transition system, then the **transition function**, $\delta^*: S\times\Sigma^* \rightarrow \wp(S)$, is defined inductively for all $s\in S$ by: $\delta^*(s, \varepsilon) = \{s\}$, and for all $x\in\Sigma^*$ and $\lambda\in\Sigma$, $\delta^*(s, x\lambda) = \bigcup_{t\in\delta^*(s,x)} \delta(t,\lambda)$.

As in the deterministic case, we will use δ to mean both δ and δ^* with the interpretation to be determined by the context. As in the deterministic case, these two functions agree on their common domain ($\delta^*(s, \lambda) = \delta(s, \lambda)$) so there is no real danger of confusion. The transition function for an NTS yields the set of all the possible state outcomes for a given input sequence. Usually we envision this as encompassing a potentially large set of possible outcomes. But, in fact, there may always be exactly one possible outcome — in this case we effectively have a deterministic transition system, so non-deterministic automata provide deterministic automata as a special case. That is, every DTS can be viewed as a NTS.

Furthermore, one of the possible sets of next states in an NTS is ∅, *none!* This provides an NTS with the ability to model a new behavior, "blocking" or stopping part way through an input sequence. Such behavior cannot be modeled with a DTS which must completely process every input sequence.

So a non-deterministic device leaves some uncertainty about how it will react to any given input. This could reflect physical device characteristics that are beyond our design control such as the temperature or humidity of the environment in which the device is placed. Non-deterministic devices may thus appear to be unrealistic or unhelpful models of computing systems, but we will indicate a little later in this section some important practical applications of these models. Irrespective of whether it is realistic, non-determinism is a valuable conceptual tool.

The concept of non-determinism is rather elusive. Computation descriptions where we do not say exactly what will happen next seem less familiar in our experience. But if, for example, we regard it as allowing us to avoid the over-specification of a forced choice between an arbitrary order in which to react to the occurrence of several "simultaneous" events when order is immaterial to the desired outcome, it may seem more natural. Our main interest is to use non-determinism as an analysis aid to ease reasoning about deterministic devices! We next see how this can be accomplished.

Definition 1.2.6: A **non-deterministic finite acceptor** (NFA) A is a 5-tuple, $A = (S, \Sigma, \delta, s_0, R)$, where (S, Σ, δ) is a non-deterministic transition system, $s_0 \in S$ is the **start state**, and $R \subseteq S$ is the set of **recognizing** (or **accepting**) states. The **language recognized** (or **accepted**) by A is $L(A) = \{x \in \Sigma^* \mid \delta(s_0, x) \cap R \neq \emptyset\}$. A language $L \subseteq \Sigma^*$ is **NFA-recognizable** if there exists a NFA A so that $L = L(A)$.

For an NFA A as in Definition 1.2.6 and $x \in \Sigma^*$, say $x = \lambda_1\lambda_2 \ldots \lambda_k$ $(k \geq 0)$ with $\lambda_i \in \Sigma$ $(1 \leq i \leq k)$, we say x has a **run** $s_0, s_1, \ldots, s_k$ in A if $s_1 \in \delta(s_0, \lambda_1)$, $s_2 \in \delta(s_1, \lambda_2)$, ... , $s_k \in \delta(s_{k-1}, \lambda_k)$. A run is **recognizing** (or **accepting**) if $s_k \in R$.

While the NFA model includes the DFA model as a special case and is still a finite description of its language, it is a distinctly different model. Not only is the NFA model different, this difference leads to a new recognition criterion. For each $x \in \Sigma^*$ there is exactly one run in a DFA, but in an NFA there may be a set of runs reflected in $\delta(s_0, x)$ — if one or more of these states is a recognizing state, then x is "recognized"; x is rejected only if *no* run is recognizing. So there are many possible behaviors associated with a single input sequence, and they may be both accepting and rejecting. Even one accepting outcome is taken to be enough for the sequence to be categorized as "recognized". Furthermore, an input sequence can be rejected either by leading only to non-recognizing states, or by not leading to any states at all (i.e., by having *no* runs) — that is by *blocking*, an entirely new computational behavior from that provided in the DFA model.

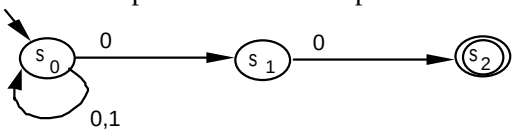
The various possible transitions of an NTS for an input are not in any way distinguished from one another — each is simply one of the conceivable outcomes. A refinement of this model assigns a probability to each of the possible next states at each step. Such devices are referred to as "probabilistic automata", and the likelihood of an outcome is used in characterizing behavior. The analysis of this model is technically more difficult and goes beyond the scope of this book — the interested reader can consult the book by Paz[Paz 71].

Example 1.2.4.

This first example of an NFA has $\Sigma = \{0,1\}$ and accepts precisely those sequences whose last two characters are '0'. We take states $S = \{s_0, s_1, s_2\}$ with s_0 as start state, and $F = \{s_2\}$. Then the next state function is given by:

δ	s_0	s_1	s_2
0	$\{s_0, s_1\}$	$\{s_2\}$	$\varnothing$
1	$\{s_0\}$	$\varnothing$	$\varnothing$

As in the case of DFAs, we normally will prefer to use state diagrams to present examples. For this example we have



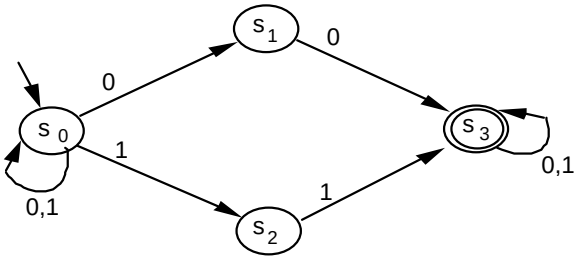
The diagram conventions used for NFAs are basically the same as those used previously for DFAs. However, an input symbol may label several edges (or none) from the same node in the non-deterministic case, but this is the only difference.

As clearly seen in the state diagram for this example, every accepting run requires that the last two symbols of the input must be '00'. Also, every such sequence clearly has an accepting run consisting of the transition from s_0 to itself for every symbol except the last two (of course, options also permit early transitions to s_1 and s_2 and subsequent blocking). In this example, non-determinism is used in every state — at s_0 there are two possible next-states when the input is '0', and in states s_1 and s_2 the absence of next-states can lead to rejection by blocking (i.e., no runs).
❑

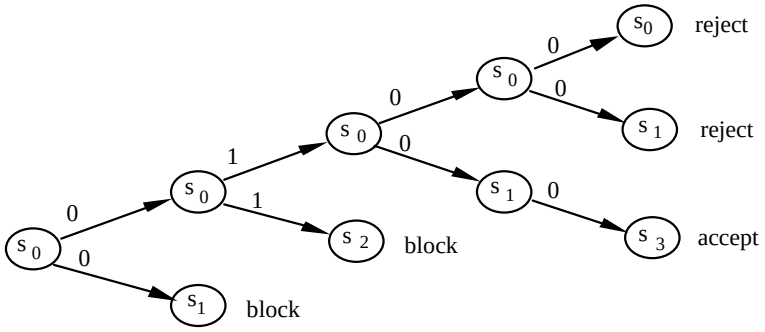
While the NFA model is apparently more complex than the DFA model, it may be that an NFA description is simpler and easier to provide than a DFA description. This can be seen in our first NFA example above — to be convinced of this, the reader should construct a DFA for the language of Example 1.2.4 (see Exercise 1.16). However, this does not imply that NFA examples are always easy to understand. In general, since there are more computational behaviors possible than with DFAs, NFAs may be even more difficult to understand. But quite often it turns out the other way.

Example 1.2.5.

The NFA below has four states, and the primary use of non-determinism occurs in the start state. With a little reflection, it will hopefully be clear to the reader what constitutes the recognized sequences.



Then for input 0100 we have the following “tree” of run possibilities.



Hence there are three runs for input 0100 (the others block before completion) with one accepting and two rejecting, and this means that 0100 is accepted.
❑

For a variety of reasons, it is occasionally of interest to refer to the connectivity structure of transition systems (e.g., see Example 1.2.6). Before proceeding with the other recognition models, we introduce three useful terms.

Definition 1.2.7: A state t of a transition system is said to be **reachable** from state s if there is $x \in \Sigma^*$ so that $\delta(s, x)=t$ (or $t \in \delta(s, x)$ in the non-deterministic case). A state is called a **generator** if every state is reachable from it. A transition system is called **strongly connected** if every state is a generator.

Among the examples we have presented so far, it can be seen that in Example 1.2.1, s_0 is a generator, but s_1 and s_2 are not. In Example 1.2.3, A_1 and A_3 are strongly connected, but A_2 is not. Note that for acceptors, there would be no loss of generality in recognition capacity if we were to assume that the start state is always a generator since deleting its unreachable states cannot effect acceptance. However, there would be no great gain either and we would sacrifice some flexibility, so we do not make such a blanket assumption (but this is the case in all our examples so far).

While we can model more computational behaviors with NFAs, it turns out that their expressive capacity is no greater than that of DFAs in the sense

of the theorem below. This result is based on the observation that at any point in a non-deterministic computation, there are only finitely many possible state outcomes so we can deterministically keep track of these possibilities.

Theorem 1.2.2: Each NFA-recognizable language is DFA-recognizable.
Proof:

This is a proof by construction. For a given NFA, we demonstrate the existence of an equivalent DFA by explicitly providing its definition. Let $A = (S, \Sigma, \delta_A, s_0, F)$ be an NFA. We define the DFA $B = (T, \Sigma, \delta_B, t_0, G)$, where $T = \wp(S)$, $t_0 = \{s_0\}$, $\delta_B(X, \lambda) = \bigcup_{s \in X} \delta_A(s, \lambda)$ for each $X \subseteq S$ and $\lambda \in \Sigma$, and $G = \{X \subseteq S \mid X \cap F \neq \varnothing\}$. Now clearly B is a well-defined DFA — its states are the subsets of S, and a unique next-state (subset) is determined for each letter in Σ. This is another instance where the abstraction of our models is helpful — states of a DFA may be anything at all, so having states that consist of subsets of states from another machine is permitted.

Claim: for each $x \in \Sigma^*$, $\delta_B(t_0, x) = \delta_A(s_0, x)$ and hence L(B) = L(A).

Proof of claim:

The proof is an induction argument on len(x).

Anchor step: len(x)=0, or $x=\varepsilon$.

The claim follows immediately from the NFA and DFA definitions.

Induction step: assume the claim is true for all x with len(x)=n and consider $x\lambda$ with $\lambda \in \Sigma$ and len(x)=n+1. Then

$\delta_B(t_0, x\lambda)$

$= \delta_B(\delta_B(t_0, x), \lambda)$ by the definition of a DFA

$= \delta_B(\delta_A(s_0, x), \lambda)$ by the induction hypothesis

$= \bigcup_{s \in \delta_A(s_0, x)} \delta_A(s, \lambda)$ by the definition of B

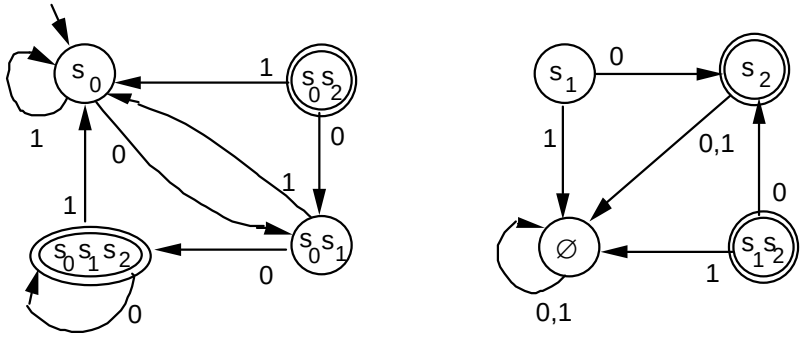
$= \delta_A(s_0, x\lambda)$ by the definition of an NFA.

This extends the induction, proving the claim. Since the claim verifies the correctness of the construction, the theorem is proven. While the claim is all we require to prove the theorem, it may be noted that the “simulation” of

NFA A that is provided by DFA B is perfect — the equality shown in the claim is true for every state of A, not just the start state.
❑

Example 1.2.6.

We illustrate the construction in the proof of Theorem 1.2.2 for the NFA of Example 1.2.4. Since the NFA in that example has 3 states, we have 8 subsets that serve as the states of the DFA. The resulting machine is given in the state diagram



Notice that only 3 of the 8 states in this DFA (i.e., $\{s_0\}$, $\{s_0, s_1\}$, and $\{s_0, s_1, s_2\}$) are essential (i.e., reachable from the start state), and the others could be deleted without changing the language accepted.
❑

The construction used in the proof of Theorem 1.2.2 is known as the "power set" construction. Analogous techniques have been found to be valuable in a number of other areas of theoretical computer science. By this construction, we see that we can always replace an N state NFA by a 2^N state DFA. Hence while NFAs and DFAs have the same expressive capacity, this proof suggests that NFAs may provide a substantial economy of expression. Of course, this proof does not foreclose the possibility that a DFA with fewer than 2^N states could be sufficient. In fact, in many cases fewer states will suffice, as seen in Example 1.2.6 — our proof guarantees a correct DFA, not an economical one. However, we will see later that this is the best upper bound possible and that there are indeed cases that require this exponential state explosion when going from an NFA to a DFA. The succinctness that can be obtained with an NFA can make it much more

understandable than the corresponding DFA so there is a significant benefit associated with the added complexity of NFAs even though expressive power has not been increased.

Note that while we do not explicitly present an algorithm for transforming an NFA to a DFA, it is clear that all elements for this have been outlined in our proof. For practical purposes, it may also be noted that one typically does not require all 2^N subsets in a corresponding DFA — only those reachable from the start state will actually be relevant, and if one "builds" a DFA beginning with the start state and only incorporating those subsets that are reached, the construction is usually substantially reduced. Instances like Example 1.2.6 are common, where fewer states than the upper bound suffice.

There is one more extension of the finite state model that will be useful. This extends the NFA model and provides for modeling still another computational behavior.

Definition 1.2.8: A **null-move (or ε-move) non-deterministic transition system** (ε-NTS) is a triple, $T = (S, \Sigma, \delta)$, where S, the **states**, and Σ, the **input alphabet**, are finite non-empty sets, and δ is a function, $\delta: S\times(\Sigma\cup\{\varepsilon\}) \rightarrow \wp(S)$, the **next-state function**.

So an ε-NTS is non-deterministic, and in addition provides the possibility of a state transition even when there is no input (i.e., the input is ε). These ε-moves can be thought of as "spontaneous transitions" that the device *may* make without external stimulus. This kind of behavior may seem rather bizarre from the perspective of our discussion to this point. However, there are computational circumstances where this behavior is both natural and essential. In constructing models of concurrent processes with shared variables, one process description does not include all the information for its potential state changes. Even if a process does nothing, its state may be changed by another process writing into a shared variable. In this context an ε-transition indicates that a shared variable is "unlocked" and available for writing by another process, and the particular ε-move will depend on the action of a cooperating process. The interested reader could consult [Hen 88] for a development along these lines. Our interest here will be to use this behavior to further ease some of our upcoming analysis, and to set the stage

for the push-down automata considered in a later chapter that utilize this ability to run without consuming input.

The ε-moves significantly complicate the ways in which transitions cascade. So before we define how an ε-NTS responds to a sequence of inputs, we must address the cascading of ε-moves.

Definition 1.2.9: for state s of an ε-NTS, **ε-closure**(s) is a set of states defined inductively by:

(i) $s \in \varepsilon\text{-closure}(s)$,

(ii) if $t \in \varepsilon\text{-closure}(s)$ and $u \in \delta(t, \varepsilon)$, then $u \in \varepsilon\text{-closure}(s)$,

(iii) nothing belongs to the ε-closure(s) unless it follows from finitely many applications of (i) and (ii).

Also, for a set of states T, $\varepsilon\text{-closure}(T) = \bigcup_{s \in T} \varepsilon\text{-closure}(s)$.

The set ε-closure(s) includes s plus all the states that are reachable from s using repeated ε-moves. A transition from s to any of these states may always occur "spontaneously" (i.e., without reading any input). With this spontaneous behavior carefully defined, we can now define the extension of the next-state function of an ε-NTS to transitions for input sequences.

Definition 1.2.10: For ε-NTS T = (S, Σ, δ), the **transition function** δ^*: $S \times \Sigma^* \to \wp(S)$ is defined inductively for each $s \in S$ by:

(i) $\delta^*(s, \varepsilon) = \varepsilon\text{-closure}(s)$,

(ii) for $x \in \Sigma^*$ and $\lambda \in \Sigma$, $\delta^*(s, x\lambda) = \varepsilon\text{-closure}\left(\bigcup_{t \in \delta^*(s,x)} \delta(t, \lambda) \right)$.

As with DTS and NTS, we use the symbol δ to denote both δ and δ^* with context serving to distinguish the intended function. However, in this case there is some chance of confusion since δ and δ^* may differ on their common domain — $\delta(s, \lambda) \subseteq \delta^*(s, \lambda)$ but they need not be equal. If clause (ii) in Definition 1.2.10 is specialized with $x = \varepsilon$, then we have

$\delta^*(s, \lambda) = \varepsilon\text{-closure}\left(\bigcup_{t \in \varepsilon\text{-closure}(s)} \delta(t, \lambda) \right)$ — ε-moves are permitted both before and after the letter λ is read.

Intuitively, when there are ε-moves, the set of possible state outcomes for an input sequence allows an arbitrary number of spontaneous transitions before and after each letter is processed. Thus

$\delta^*(s, \lambda_1) = \varepsilon\text{-closure}(\{t \mid \exists r \in \varepsilon\text{-closure}(s)) \text{ with } t \in \delta(r, \lambda_1) \})$,

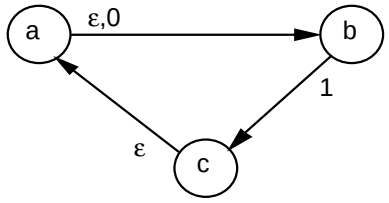
$\delta^*(s, \lambda_1\lambda_2) = \varepsilon\text{-closure}(\{t \mid \exists r \in \delta^*(s, \lambda_1) \text{ with } t \in \delta(r, \lambda_2)\})$

etc.

Example 1.2.7.

For the simple ε-NTS below we have: ε-closure(a) = {a, b}, ε-closure(b) = {b}, and ε-closure(c) = {a, b, c}. Therefore inputs of length one (letters) incorporate numerous possible ε-moves into the determination of the transitions. So, for instance, $\delta^*(a, 1) = \{a, b, c\}$ since starting from state a

- state a is reached by: ε-move, 1-move, ε-move;
- state b is reached by: ε-move, 1-move, ε-move, ε-move;
- state c is reached by: ε-move, 1-move.



❑

Definition 1.2.11: A **null-move non-deterministic finite acceptor** (ε-NFA) A is a 5-tuple, $A = (S, \Sigma, \delta, s_0, R)$, where (S, Σ, δ) is a null-move non-deterministic transition system, $s_0 \in S$ is the **start state**, and $R \subseteq S$ is the set of **recognizing** (or **accepting**) states. The **language recognized** (or **accepted**) by A is $L(A) = \{x \in \Sigma^* \mid \delta(s_0, x) \cap R \neq \varnothing\}$. A language $L \subseteq \Sigma^*$ is **ε-NFA-recognizable** if there exists an ε-NFA A so that L = L(A).

Of course, every NFA is an ε-NFA, simply one where the set of ε-moves happens to be empty in every case (i.e., for all s, $\delta(s, \varepsilon) = \varnothing$). Hence we have properly extended the computations that can be modeled. But while we again have a proper extension of the computational behaviors that can be modeled, this still does not increase the recognition capacity as we next prove.

Lemma 1.2.3: if X and Y are any subset of states of an ε-NFA, then
(a) ε-closure(ε-closure(X)) = ε-closure(X), and
(b) ε-closure(X ∪ Y) = ε-closure(X) ∪ ε-closure(Y).
The proof of this lemma is left as an exercise.

Theorem 1.2.4: Each ε-NFA-recognizable language is NFA-recognizable.
Proof:

This proof is by construction. Given an ε-NFA $A = (S, \Sigma, \delta_A, s_0, R_A)$, we provide an explicit construction to show the existence of an NFA B so that L(A) = L(B). Define $B = (S, \Sigma, \delta_B, s_0, R_B)$, where $\delta_B(s, \lambda) = \delta^*_A(s, \lambda)$ for all $s \in S$ and $\lambda \in \Sigma$, and

$$R_B = \begin{cases} R_A \cup \{s_0\} \text{ -- if } \varepsilon\text{-closure}(s_0) \cap R_A \neq \varnothing \\ R_A \text{ -- otherwise.} \end{cases}$$

The idea of this construction is that the next-state function of B is expanded to include all the states that can result from ε-moves in A into the possible state outcomes for each individual letter. Then the same transitions are possible in B without the ε-moves. Clearly by the definition of R_B, $\varepsilon \in L(A)$ if and only if $\varepsilon \in L(B)$; if ε-closure$(s_0) \cap R_A \neq \varnothing$, then adding s_0 as an accepting state would not change the language recognized. For non-null strings we make the

Claim: for each $s \in S$ and $x \in \Sigma^*$ with $len(x) \geq 1$, $\delta^*_A(s, x) = \delta^*_B(s, x)$, and hence L(A) = L(B).
Proof of claim:

The proof of this claim is by induction on len(x).
Anchor step: for len(x) = 1, $x = \lambda \in \Sigma$ and the claim follows immediately from the definition of δ_B.
Induction step: assume the claim is true for all x with len(x) = n and consider an input of length n+1, say xλ, where $\lambda \in \Sigma$. Then

$$\delta^*_B(s, x\lambda)$$

$$= \bigcup_{t \in \delta^*_B(s,x)} \delta_B(t, \lambda) \qquad \text{by the definition of NFA transitions}$$

$$= \bigcup_{t \in \delta^*_A(s,x)} \delta_B(t, \lambda) \qquad \text{by the induction hypothesis}$$

$= \bigcup_{t \in \delta^*_A(s,x)} \delta^*_A(t, \lambda)$ by the definition of δ_B

$= \bigcup_{t \in \delta^*_A(s,x)} \varepsilon\text{-closure}(\delta^*_A(t, \lambda))$ by Lemma 1.2.3(a) and $\delta^*_A(t, \lambda)$

$= \varepsilon\text{-closure}(\bigcup_{t \in \delta^*_A(s,x)} \delta^*_A(t, \lambda))$ by Lemma 1.2.3(b)

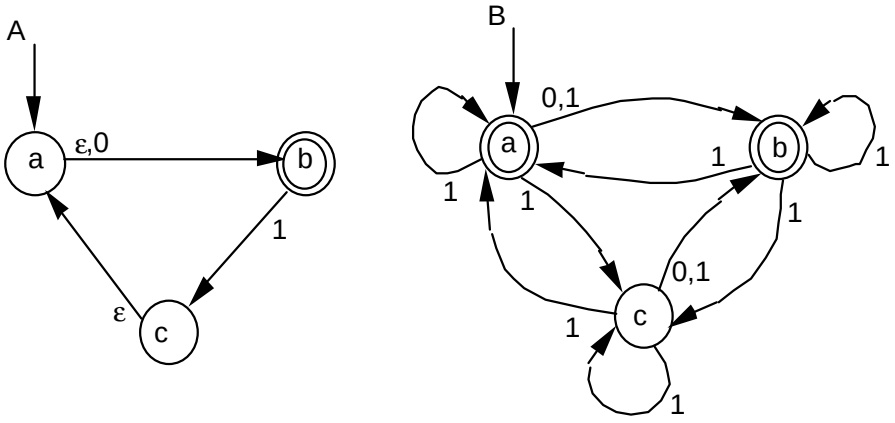
$= \delta^*_A(s, x\lambda)$ by the definition of ε-NFA.

This extends the induction proving the claim, and completing the proof of Theorem 1.2.4.
❑

We see in the proof of Theorem 1.2.4 that an ε-NFA is equivalent to an ordinary NFA with the same number of states, so there is not the economy of states as in the NFA versus DFA case. However, the number of atomic transitions (i.e., edges) defined in the NFA may expand greatly over those in the ε-NFA. Consequently, although there is no increase in the recognition capacity, ε-NFAs may be helpful in achieving brevity and clarity.

Example 1.2.8.

If we follow the construction in the proof of Theorem 1.2.4 for the deceptively simple ε-NFA A (see Example 1.2.7), we obtain the NFA B as shown below.



Is it clear that these two acceptors recognize the same language? What is this language?
❑

This section has developed a collection of basic models of computation. The computations are very simple — process a string symbol by symbol and determine whether or not it belongs to a designated language. We saw that DFAs, NFAs and ε-NFAs all have the same recognition capacity, even though each successive mechanism manifests additional computational behavior (e.g., blocking, and spontaneous state change). Non-determinism might have initially seemed artificial and somewhat mysterious, but the natural DFA simulation removes the perplexity. DFAs have the simplest theoretical basis, but the explosion from an N-state NFA to a 2^N-state DFA can make DFAs comparatively impractical. We will see that ε-NFAs aid in the systematic construction of acceptors, but the ε-moves can lead to subtle complication, and are a difficulty in algorithm realization. Through implementations that simulate the power set construction, NFAs have the greatest accepted practical roles in search string processors in text editors, lexical analyzers in compilers, and for modeling communication protocols.

Section 1.3: Equality of regular expressions and acceptors

In the previous section we have seen a series of acceptor mechanisms that provide for modeling various computational phenomena. But all three of them achieve exactly the same expressive capacity when measured by the sets of sequences that can be recognized. The symbol by symbol processing of one specific string by the recognizer models contrasts with the approach of generating the set of member sequences provided by regular expressions of section 1. In this section we demonstrate that the descriptive capacity of these recognition models is exactly the same as that of the regular expressions.

Theorem 1.3.1: Every DFA-recognizable language is regular.
Proof:

This is a constructive proof where we show how to form a regular expression that describes exactly the recognizable sequences. Without loss of generality, we assume that DFA A = (S, Σ, δ, s_1, R), where S = {s_1, s_2, … ,

$s_n\}$ for some $n \geq 1$. That is, we arbitrarily assign some sequential order to the states of A. This ordering will provide the basis for an inductive development of a regular expression. While any ordering will do, different orderings will usually result in different (but equivalent) regular expressions. For $x \in \Sigma^*$, we will say that the transition $\delta(s_p, x) = s_q$ "passes through" state s_r if for some $y,z \in \Sigma^+$, $x = yz$ and $\delta(s_p, y) = s_r$. Informally, an input sequence "passes through" only the intermediate states of the run, not the beginning and ending states. For $1 \leq p,q \leq n$ and $0 \leq r \leq n$ we define a collection of sets L_{pq}^{r} so that $x \in L_{pq}^{r}$ if and only if x takes s_p to s_q without "passing through" any state s_m with $m > r$. Formally

$$L_{pq}^{r} = \{x \in \Sigma^* \mid \delta(s_p, x) = s_q \text{ and if } x=yz \text{ with } y,z \in \Sigma^+ \text{ and } \delta(s_p, y) = s_m, \text{ then } m \leq r\}.$$

Then $L_{pq}^{n} = \{x \in \Sigma^* \mid \delta(s_p, x) = s_q\}$ and $L(A) = \bigcup_{s_q \in R} L_{1q}^{n}$. We need only show that each of the languages L_{pq}^{r} is regular, and regular expressions for them are immediately combined to obtain one for L(A). Strings in L_{pq}^{0} pass through no states, and hence these languages are easily determined. Thus

$$L_{pq}^{0} = \begin{cases} \{\lambda \in \Sigma \mid \delta(s_p, \lambda) = s_q\} \text{ if } p \neq q \\ \{\varepsilon\} \cup \{\lambda \in \Sigma \mid \delta(s_p, \lambda) = s_q\} \text{ if } p = q \end{cases}.$$

Since these are finite (or empty) sets, they are regular and have simple regular expressions. And for $r>0$ we have the following fundamental identity:

$$(\S)\ L_{pq}^{r} = L_{pq}^{r-1} \cup L_{pr}^{r-1} (L_{rr}^{r-1})^* L_{rq}^{r-1}.$$

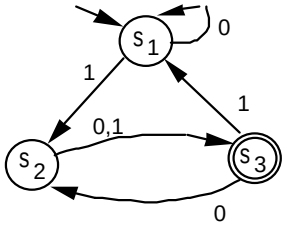
The identity (§) can be readily verified. The sequences in L_{pq}^{r} determine transitions from s_p to s_q which pass only through states numbered r or smaller. Now if such a sequence does not pass through s_r at all, then it belongs to L_{pq}^{r-1}. If a sequence $x \in L_{pq}^{r}$ does pass through s_r one or more

times, then we identify the first and last times that x passes through s_r, and x=uvw where $\delta(s_p, u) = s_r$ and this is the first encounter, $\delta(s_r, w) = s_q$ and this is the last encounter (and could be the same as the first), and $\delta(s_r,v) = s_r$. Clearly $u \in L_{pr}^{r-1}$ and $w \in L_{rq}^{r-1}$ since s_r is not passed through by these sequences. Also v may repeatedly (zero or more times) pass through s_r, but the subsequence between one encounter and the next is in L_{rr}^{r-1}, so $v \in (L_{rr}^{r-1})^*$. Clearly (§) provides the essence of an induction which shows that all the languages L_{pq}^{r} are regular, and in fact, we can systematically develop regular expressions from those for L_{pq}^{0} using (§).

❑

In fact, the proof strategy used in this theorem effectively provides a systematic algorithm for transforming a DFA description into a regular expression description of the same language. It is informative to apply the method of proof of Theorem 1.2.1 to a specific DFA to re-express the language of the DFA by a regular expression. It will be noted that while this is an infallible method of obtaining a regular expression from a DFA, it usually yields a highly "non-optimal" (i.e., overly complex) result.

Example 1.3.1.



$L(A) = L_{13}^{3}$

Just from inspection of the state diagram, we see that $L_{13}^{0} = \mu(\varnothing)$, and $L_{11}^{0} = \mu(\varepsilon+0)$. Therefore from (§) $L_{13}^{1} = L_{13}^{0} \cup L_{11}^{0} (L_{11}^{0})^* L_{13}^{0} = \mu(\varnothing + (\varepsilon+0)(\varepsilon+0)^* \varnothing)$. This already illustrates the disadvantage of following this procedure in a completely mechanical way. The regular expression $\varnothing +$

$(\varepsilon+0)\,(\varepsilon+0)^*\,\varnothing$ correctly describes L_{13}^{1}, however $(\varnothing + (\varepsilon+0)(\varepsilon+0)^*\,\varnothing) \equiv \varnothing$, and $\varnothing$ much more succinctly and clearly indicates the appropriate result (it's clear from the state diagram that there is no transition from s_1 to s_3 that passes only through s_1). Such obvious simplifications should be made when applying (§) to specific examples, and even then excessively messy results are usually produced.

Similarly from (§) $L_{13}^{2} = L_{13}^{1} \cup L_{12}^{1} (L_{22}^{1})^* L_{23}^{1}$, and since we can readily determine that $L_{12}^{1} = \mu(0^*1)$, $L_{22}^{1} = \mu(\varepsilon)$, and $L_{23}^{1} = \mu(0+1)$ we have that $L_{13}^{2} = \mu(0^*1(0+1))$.

The reader should complete this example (see Exercise 1.46) to obtain a regular expression for $L_{13}^{3} = L(A)$ and take some time to compare these two descriptions.
❑

Our first result in this section has shown us that the regular expression mechanism is at least powerful enough to express any language that can be described by any acceptor. But in fact, we next show that the converse result is also true so that these two quite different mechanisms have equivalent descriptive capacity.

Theorem 1.3.2: Each regular language is ε-NFA-recognizable, and hence by Theorems 1.2.4 and 1.2.1 is also NFA-recognizable and DFA-recognizable. In fact, we prove that the ε-NFA can always be chosen to have a single accepting state.
Proof:

This proof is again a structural induction proof. Since for any regular language, we can rely on a regular expression description, and we use the structure of the regular expression to guide the analysis. In particular, let $L = \mu(\alpha)$ for regular expression α, and the induction is on the number of operations in description α. Based on α, we define ε-NFA A.

Anchor step: α contains no operations

subcase 1: $\alpha = \varepsilon$

In this case we can recognize L = {ε} with one state that is both initial and final, and where all next-states are null. A = ($\{s_0\}$, Σ, δ, s_0, $\{s_0\}$), where $\delta(s_0, \lambda) = \varnothing$ for all $\lambda \in \Sigma$.

subcase 2: $\alpha = \varnothing$

In this case we can recognize L = ∅ with a two state ε-NFA — one initial state, one final state, and no transitions. A = ($\{s_0, s_1\}$, Σ, δ, s_0, $\{s_1\}$), where $\delta(s_i, \lambda) = \varnothing$ for all $\lambda \in \Sigma$, and i=1,2.

subcase 3: $\alpha = \lambda \in \Sigma$

In this case we can recognize L = {λ} with a two state ε-NFA — one initial state, one final state, and one transition. A = ($\{s_0, s_1\}$, Σ, δ, s_0, $\{s_1\}$), where $\delta(s_0, \lambda) = \{s_1\}$, and $\delta(s_i, \lambda') = \varnothing$ for all $\lambda' \in \Sigma$, $\lambda' + \lambda$, otherwise.

Induction step: assume the theorem is true for all regular expressions with $n \geq 0$ or fewer operations, and consider regular expression α with n+1 operations.

subcase 1: $\alpha = \alpha_1 + \alpha_2$

So α_1 and α_2 each involve n or fewer regular expression operations. Thus, the induction hypothesis implies that there exist ε-NFAs A_i with $\mu(\alpha_i) = L(A_i)$, i=1,2. We easily stitch together A_1 and A_2 as submachines to create an acceptor for μ(α). We just enable ε-moves to start (non-deterministically) either A_1 or A_2 and when one of these submachines finishes (i.e., enters one of its accept states), it has an added ε-move option to exit to the accept state for the top-level. Formally, let $A_i = (S_i, \Sigma, \delta_i, s_0^i, \{r^i\})$ for i=1,2, and without loss of generality assume that $S_1 \cap S_2 = \varnothing$. Then introduce new abstract symbols $\{s_0, r\}$ not occurring in A_1 or A_2 and define A = ($\{s_0, r\} \cup S_1 \cup S_2$, Σ, δ, s_0, {r}), where

$\delta(s_0, \varepsilon) = \{s_0^1, s_0^2\}$,

$\delta(t, \varepsilon) = \{r\} \cup \delta_1(t, \varepsilon)$ for all $t \in R_1$,

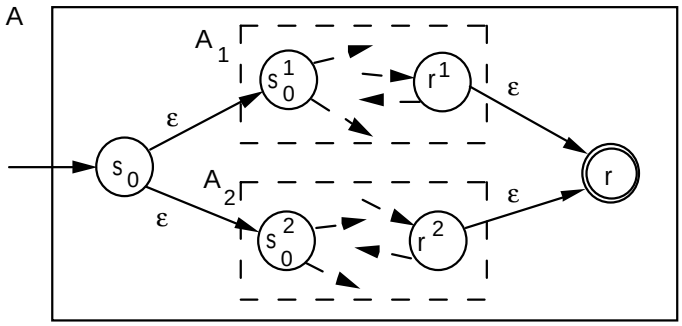
$\delta(t, \lambda) = \delta_1(t, \lambda)$ for all $t \in S_1$ and $\lambda \in \Sigma$,

$\delta(t, \varepsilon) = \{r\} \cup \delta_2(t, \varepsilon)$ for all $t \in R_2$,

$\delta(t, \lambda) = \delta 2(t, \lambda)$ for all $t \in S_2$ and $\lambda \in \Sigma$,

and $\delta(s, \lambda) = \varnothing$ in all other cases.

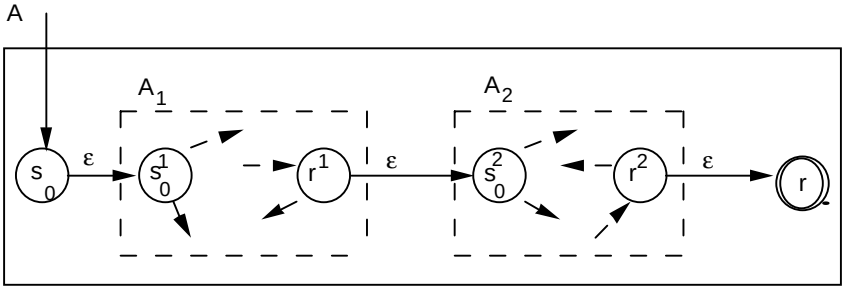
Schematically we can depict this construction of recognizer A by the diagram below.



It should be clear that the accepting runs in A are just those of A_1 together with those of A_2 so that $L(A) = L(A_1) \cup L(A_2) = \mu(\alpha_1+\alpha_2)$. Details of this analysis are omitted.

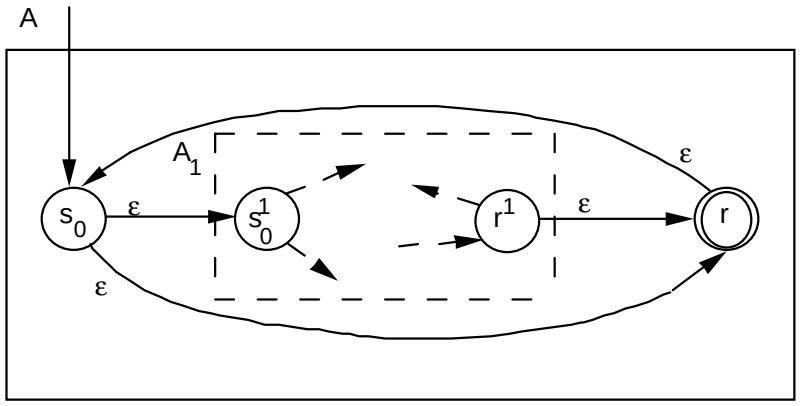
subcase 2: $\alpha = \alpha_1 \bullet \alpha_2$

A construction analogous to that in the first subcase also applies here. Below is the schematic depiction of the construction — details similar to those in the first case are left to the reader.



subcase 3: $\alpha = (\alpha_1)^*$

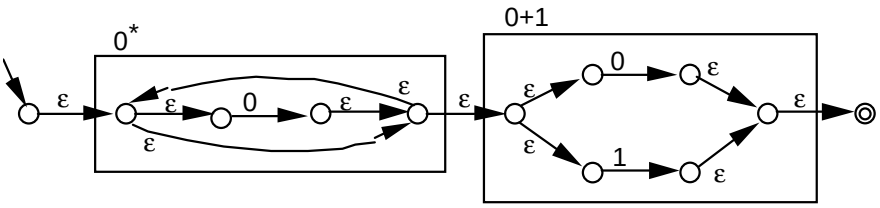
Again a construction analogous to that in the first subcase applies here. Below is the schematic depiction of the construction — details similar to those in the first case are left to the reader.



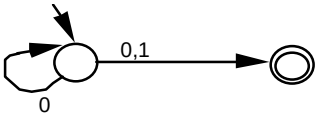
❑

The details of the constructions indicated in Theorem 1.3.2 are subject to a number of minor variations. In particular, it is not difficult to avoid adding all the new states indicated in the constructions shown here. This can be of some benefit if one chooses to mechanically follow the construction outlined here to transform a regular expression into a recognizer. We have emphasized clarity here rather than economy. While this analysis effectively provides an algorithm to translate a regular expression to an ε-NFA, it suffers from a similar practical difficulty to that noted previously for the algorithm that converts in the opposite direction. Namely, if one mechanically follows the indicated procedure, the resulting ε-NFA is guaranteed to be correct, but in almost all cases the result is a highly non-optimal recognizer.

Example 1.3.2.
If we mechanically apply the construction of Theorem 1.3.2 to the regular expression 0*•(0+1), the reader should verify that we obtain the 12 state acceptor



If we employ a little intuition and insight to merging redundant states, we can develop a recognizer with only two states.



❑

The unification of the regular and the acceptor families of languages shown by Theorems 1.3.1 and 1.3.2 permits us to collectively affirm all the properties previously established for either family. Furthermore, by combining these properties we can uncover further properties that are not readily apparent using either description method alone.

Corollary 1.3.3: the regular languages are closed under intersection.
Proof:

By Theorem 1.3.2, each regular language is ε-NFA-recognizable. Then by Theorems 1.2.4 and 1.2.2, it is DFA-recognizable. Hence by Theorem 1.2.1, its complement is DFA-recognizable and hence regular. From DeMorgan's laws, for any sets $X \cap Y = \neg(\neg X \cup \neg Y)$, so from closure under union and complement we can infer closure under intersection.

❑

One additional frequently useful operation on strings is to reverse them end for end, that is, just write them backwards. This will cause characteristics not occurring until the end to be present right at the start and can simplify some processes.

Definition 1.3.1: the **reversal** of a string $x \in \Sigma^*$, written x^R, is defined inductively as follows: $\varepsilon^R = \varepsilon$, and for each $\lambda \in \Sigma$ and $x \in \Sigma^*$, $(\lambda x)^R = x^R \lambda$.

Also, for a language $L \subseteq \Sigma^*$, the **language reversal**, L^R, is the element-wise application, $L^R = \{x^R \mid x \in L\}$.

Thus, for example, $(0^*1)^R = 10^*$ and $((001)^*)^R = ((001)^R)^* = (100)^*$. And the reversal operation is regularity preserving in general as we show next.

Theorem 1.3.4: for each regular language $L \subseteq \Sigma^*$, L^R is also regular.
Proof:

By Theorem 1.3.2 we can assume that L = L(A) where A is an ε-NFA with one final state, say A = (S, Σ, δ, s_0, {r}). Then define ε-NFA A' = (S, Σ, δ', r, {s_0}), where for all $s \in S$ and $\sigma \in \Sigma \cup \{\varepsilon\}$, $\delta'(s, \sigma) = \{t \mid s \in \delta(t, \sigma)\}$. Then the idea is that A' just simulates A "running backwards". That is, there is a transition from state s to state t in A' exactly when there is a transition from state t to state s in A. Hence whenever A has a run from s_0 to r, A' has a run from r to s_0, and vice-versa. The reader is left to verify the details. But since this is true, $L(A') = L^R$ and so by Theorems 1.2.4, 1.2.2, and 1.3.1 this result is proven.
❑

The construction in the proof of Theorem 1.3.4 graphically reveals the descriptive power of non-determinism. From Theorems 1.2.2 and 1.2.4, we know that a DFA for L^R must exist. But while the ε-NFA description is practically immediate, a direct description of the DFA would be highly challenging to devise and difficult to verify.

While superficially the regular expressions do not appear to describe "computations", we have seen in this section that such a view of them is available just below the surface. The regular expressions provide a generative orientation to describing languages that is fully equivalent to each of the models of acceptors that provide a recognition orientation to languages. Moreover, we have effectively presented algorithms (though not especially practical ones) for translating from any of the descriptions to any other. From this point on we are free to use whichever of the descriptive mechanisms that have been introduced that most readily suits the circumstances of the moment. The constructions presented in this section have revealed the strong similarity between Kleene closure in regular

expressions and cycles in recognizers — nesting of either corresponds to nesting of the other. Also, language concatenation is analogous to sequential execution, and union resembles branching, an analogy we will pursue further in Section 2.4.

Section 1.4: Two-way acceptors‡

In section 2 of this chapter, we investigated several different models of acceptors. These models allowed us to reproduce formal counterparts for a variety of computational behaviors, although in the end all the models yielded the same recognition capacity. One common feature of those models was that they all provided for unidirectional scanning of the input sequence. The finite-state character of the memory meant that for inputs of unbounded (but finite) length, only limited partial information about the previously scanned subsequence could be retained. One might surmise that the ability to re-scan past input would provide a means to overcome this limitation and lead to a model that provides greater recognition capability. In this section we formalize the exploration of this intuition.

To pursue this idea, our first step is to provide a formal model of an acceptor that is not bound to process the input sequence in a unidirectional fashion. We do this by permitting the acceptor to process symbols in either the forward direction (i.e., left-to-right), or to (re)process symbols in the reverse direction (right-to-left). Thus any portion of the input sequence can be processed repeatedly and decisions to re-scan can be based on symbols which come later in the sequence.

Definition 1.4.1: a **two-way deterministic finite acceptor** (2W-DFA) A is a 5-tuple, $A = (S, \Sigma, \delta, s_0, R)$, where S, Σ, s_0, and R are as in a DFA, and the next-state function δ: $S\times\Sigma \rightarrow S\times\{\boldsymbol{L}, \boldsymbol{R}\}$, where ***L*** (for *left*) and ***R*** (for *right*) are required to be two abstract symbols otherwise not appearing in A.

So if the "next-state" function defines $\delta(s, \lambda) = (t, \boldsymbol{R})$, when the acceptor is in state s and scanning the letter λ, it changes to state t and next scans the input letter immediately to the right of λ in the input sequence — exactly as

‡ This section can be omitted without loss of continuity.

in a DFA. The new idea is that if the "next-state" function defines $\delta(s, \lambda) = (t, \boldsymbol{L})$, when the acceptor is in state s and scanning the letter λ, it changes to state t and next scans the input letter immediately to the *left* of λ in the input. Thus there is complete flexibility in the scanning direction to be used at any point.

Now with 2W-DFAs, extending from these atomic next-state moves to the transitions on an input sequence becomes more involved than in the one-way case. The letters that have been previously scanned cannot be eliminated from further consideration, since they may be re-scanned again later. To analyze these bi-directional computational behaviors, we introduce some additional concepts.

Definition 1.4.2: for 2W-DFA $A = (S, \Sigma, \delta, s_0, R)$, an **instantaneous description** (ID) is a sequence belonging to the set $\Sigma^* \bullet S \bullet \Sigma^*$.

The idea of an ID xsy (where $x,y \in \Sigma^*$ and $s \in S$) is that it provides a snapshot of *all* of the information about the computation of the 2W-DFA at a given moment, namely:

(1) its current state (s),
(2) the entire input sequence (xy), and
(3) the letter currently being scanned (the first letter of y), including the position in the input.

With this information, we can always determine exactly how the computation will continue.

Definition 1.4.3: given 2W-DFA $A = (S, \Sigma, \delta, s_0, R)$, we say IDs I_1 and I_2 determine a **run step**, written $I_1 \vdash I_2$, if $I_1 = \lambda_1\lambda_2 \ldots \lambda_{p-1}s\lambda_p \ldots \lambda_n$ $(1 \le p \le n)$ and

either $\delta(s, \lambda_p) = (t, \boldsymbol{R})$, and

$I_2 = \lambda_1\lambda_2 \ldots \lambda_{p-1}\lambda_p t\lambda_{p+1} \ldots \lambda_n$ ($\lambda_{p+1} \ldots \lambda_n = \varepsilon$ for p=n),

or $p>1$, $\delta(s, \lambda_p) = (t, \boldsymbol{L})$, and

$I_2 = \lambda_1\lambda_2 \ldots \lambda_{p-2}t\lambda_{p-1}\lambda_p \ldots \lambda_n$ ($\lambda_1 \ldots \lambda_{p-2} = \varepsilon$ for p=2).

IDs I_1 and I_2 determine a (k-step) **run**, written $I_1 \overset{*}{\vdash} I_2$, if there exist IDs J_0, J_1, … , J_k $(k \ge 0)$ so that $I_1 = J_0$, $I_2 = J_k$, and $I_i \vdash I_{i+1}$ $(0 \le i < k)$.

We have defined runs in 2W-DFAs so that when the 2W-DFA moves right on the rightmost letter of the input, no subsequent run steps are possible; also a run step moving left when the 2W-DFA is on the leftmost letter of the input is impossible, and this also terminates a run. Since this model is deterministic, the next-state function is required to be defined for all possible combinations, and no blocking on the "interior" of the input sequence can occur. However, a new computational behavior is now possible — an infinite loop can occur with the acceptor indefinitely moving back and forth, first in one direction and then the other.

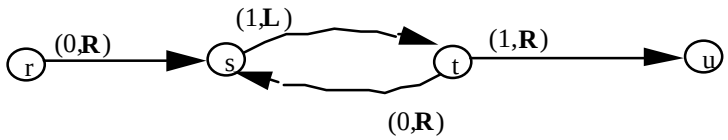
Since we now have additional computational behaviors, we need to carefully consider the conditions that we wish to employ in order that a given input be distinguished as "recognized".

Definition 1.4.4: for 2W-DFA $A = (S, \Sigma, \delta, s_0, R)$, the set of sequences **recognized** (or **accepted**) is $L(A) = \{x \in \Sigma^* \mid s_0 x \vdash^* xt \text{ and } t \in R\}$.

So the criteria we choose is for the acceptor to begin on the leftmost letter in its start state, move off the right of the input (hence guaranteeing the input has been scanned in its entirety), and terminate in an accept state. Other behaviors (i.e., moving off the right in a non-accept state, moving off the left in any state, or infinite looping) are understood to reject the input.

Observe that an ordinary DFA can be regarded as a special case of a 2W-DFA that just always moves right. The acceptance criteria for DFAs matches the criteria adopted here for 2W-DFAs, so all regular sets are immediately seen as recognized by 2W-DFAs as well. Of course, the original question was whether this new capability to make "multiple passes" over the input provides a *greater* recognition capability. Before proceeding with the answer to this question, we will consider a couple of examples of 2W-DFAs.

Note that we could still use state diagrams to present examples of 2W-DFAs by adding the direction of the move to the labels of edges. However, we do not do this for 2W-DFAs since it tends to be misleading because the added direction of scan breaks the connection between paths and transitions. For instance, consider the (partial) state diagram below.



It appears that there is a transition from state r to state u. But if we look more closely at the runs, we find that r011 |— 0s11 |— t011 |— 0s11 |— … (infinite loop) and, in fact, state u can never be reached! Therefore for 2W-DFA examples we will use the tabular form of presentation of next-state functions.

Example 1.4.1

For this 2W-DFA Σ = {0, 1}, and S = {a, b, c} with start state a, and accepting states {a}. The next-state function is given by the table below.

δ	a	b	c
0	(a,***R***)	(c,***R***)	(c,***R***)
1	(b,***L***)	(b,***L***)	(a,***R***)

So this table indicates, for instance, that δ(a, 0) = (a, ***R***), etc. A few examples of its computations are as follows:

a001 |— 0a01 |— 00a1 |— 0b01 |— 00c0 |— 001a — accept,

a10 |— — halt/reject,

a011 |— 0a11 |— b011 |— 0c11 |— 0a11 |— … — infinite loop/reject.

More generally, we see that any input starting with '1' is rejected, so all non-null recognized strings must begin with '0'. But not all sequences starting with '0' are recognized as we've noticed above. For strings of the form 0^+1, we have a0^+1 $\vdash^*$ 0^+1a, while a0^+11 $\vdash^*$ 0^+1a1 $\vdash^*$ 0^+1a1 — infinite loop. Hence the recognized language by this 2W-DFA is $(0^+1)^* 0^*$. ❑

The next example provides a more complicated 2W-DFA that further illustrates how much difficulty the bi-directional scan can add to understanding the nature of the computations that are described.

Example 1.4.2

For this 2W-DFA S = {0, 1}, and S = {a, b, c, d, e} with start state a, and accepting states {e}. The next-state function is given by the table below.

δ	a	b	c	d	e
0	(a,***R***)	(a,***R***)	(d,***L***)	(e,***R***)	(e,***R***)
1	(b,***R***)	(c,***L***)	(c,***L***)	(c,***L***)	(e,***R***)

For this 2W-DFA the input 0011 yields the run:

a0011 |— 0a011 |— 00a11 |— 001b1 |— 00c11 |— 0c011 |— d0011 |— 0e011 |— 00e11 |— 001e1 |— 0011e — accept,

and input 1100 yields the run: a1100 |— 1b100 |— c1100 — reject. What language is recognized by this 2W-DFA?

❑

The following result shows that while adding bi-directional scanning leads to much more complicated computations, it does not increase recognition capacity. This was considered quite surprising when first discovered by Rabin and Scott in an early landmark study [R-S 59], and this conclusion remains as one of the most difficult claims to justify in finite automata theory. The basic difficulty is that to simulate a 2-way DFA with a 1-way DFA, while things are obvious for moves to the right, when the 2-way machine moves left, the 1-way machine can only continue moving right! So the 1-way DFA must record sufficient information the first time it sees the input (i.e., have enough states) to enable this. Since perfect recall is impossible (the length of inputs has no bound, so any finite number of states will have its capacity exceeded), the question is: *what* to "remember"? That depends on what the 2-way DFA is going to look at when it starts moving left, so the 2-way transitions must be the guide.

Theorem 1.4.1: each language accepted by a 2W-DFA is regular.
Proof:

Our proof of this theorem is a construction. Given a 2W-DFA $A = (S, \Sigma, \delta, s_0, R)$, we define an ordinary DFA A' that recognizes the same language. To accomplish this we need to determine how to "simulate" the bi-directional scanning of a 2W-DFA while using a unidirectional scan. This is apparently impossible, and can only be indirectly accomplished if all the information that A will *ever* extract on its potentially unlimited number of passes over an input can be extracted and stored in finite state memory during a single scan of the input. This is the strategy of the construction developed here.

The first step in this construction is to determine a set that will suffice for the states of A'. The set used here is motivated by the following observations. Suppose that we are considering an input $x\lambda$, and A' has successfully simulated A's computation on x and then encounters $\lambda \in \Sigma$. If A

moves right on λ, then A' can directly simulate this. But if A moves left when reading λ, then the unidirectional nature of A' requires it to keep moving right! Since A moves left onto x, the DFA A' can only indirectly simulate this by "forecasting" A's behavior and directly proceeding as A eventually will! The transition to make in the xλ computation requires A' to have knowledge of what A does when it is started in a given state on the *rightmost* letter of x!

We are going to record the necessary knowledge about computations of A on inputs $x \in \Sigma^*$ in the form of functions that supply the critical information when given a suitable argument. To this end we first introduce a new abstract symbol, say θ, where $\theta \notin S$. With each $x \in \Sigma^*$, we associate function δ_x: $S \cup \{\theta\} \rightarrow S \cup \{\theta\}$ defined by the following:

(a) for $s \in S$

$$\delta_x(s) = \begin{cases} t \text{ -- if } x=y\lambda \text{ with } \lambda \in \Sigma \text{ and } ys\lambda \vdash^* y\lambda t \\ \theta \text{ -- otherwise} \end{cases}$$

(b)

$$\delta_x(\theta) = \begin{cases} t \text{ -- if } s_0 x \vdash^* xt \\ \theta \text{ -- otherwise} \end{cases}$$

The information about the behavior of A on input x that is "recorded" in function δ_x should be carefully noted. The symbol θ is just an added abstract symbol — we need one more than there are states in the 2W-DFA. It plays two roles in the function descriptions. As a function result, it denotes that the corresponding run of the 2W-DFA is "unproductive" (i.e., never gets the machine back to the symbol at the right of where it started). As a function argument, it gives one more place to "hang" needed information, namely, for runs that start on the leftmost (instead of rightmost) symbol in the start state (instead of a parameter state), what state they end in off the right (or θ if unproductive). When A is started on the *rightmost* letter of x in state s, $\delta_x(s) = \theta$ denotes that A never moves back off the right of x (A may move off the

left of x, or may loop on x); $\delta_x(s) = t$ denotes that A does move off the right of x and is in state t when this happens. When A is started on the *leftmost* symbol of x in state s_0, $\delta_x(\theta) = \theta$ denotes that A never moves off the right of x; $\delta_x(\theta) = t$ denotes that A does move off the right of x and is in state t when this happens. Also note that with these definitions $\delta_\varepsilon(s) = \theta$ for all states s (since ε has no rightmost symbol), and $\delta_\varepsilon(\theta) = s_0$ (since A has a run off the right with zero steps).

While there are infinitely many sequences $x \in \Sigma^*$, there are only finitely many different functions δ_x. This is true since a function is determined by the result it yields for each argument, and there are finitely many possible outcomes in $S \cup \{\theta\}$ for each of the finitely many arguments in $S \cup \{\theta\}$. Again we take advantage of the abstractness of our models that permit any finite set to serve as states, and it is this finite collection of functions that we take for the state set of the DFA A', namely $S' = \{\delta_x \mid x \in \Sigma^*\}$. Particularly notice that the functions themselves are quite distinct from their "names" in Σ^*. That is, two different sequences $x,y \in \Sigma^*$ may "name" the same function (i.e., $\delta_x = \delta_y$), so that these functions have possibly infinitely many "aliases". Indeed, the fact that Σ^* is infinite, and the number of different functions is finite, guarantees the existence of such cases.

Now we can provide the definition of the DFA. We take $A' = (S', \Sigma, \delta', \delta_\varepsilon, R')$, where $R' = \{\delta_x \mid \delta_x(\theta) \in R\}$, and $\delta'(\delta_x, \lambda) = \delta_{x\lambda}$.

By definition $x \in L(A')$ if and only if $\delta'(\delta_\varepsilon, x) = \delta_{\varepsilon x} = \delta_x \in R'$. But $\delta_x \in R'$ if and only if $\delta_x(\theta) \in R$, and $\delta_x(\theta) \in R$ if and only if $x \in L(A)$. Therefore $L(A') = L(A)$.

Our proof is complete, except for one rather subtle, but very significant, gap. In the definition of the next-state function of A', $\delta'(\delta_x, \lambda) = \delta_{x\lambda}$, we have relied on an unspecified choice x among the many potential aliases for the function δ_x. But this next-state must be a single well determined function — it can't vary with different choices of alias. This leaves a question of

whether we have a coherent definition for δ' since if $\delta_x = \delta_y$, we could just as well say that $\delta'(\delta_x, \lambda) = \delta'(\delta_y, \lambda) = \delta_{y\lambda}$, a possibly different outcome. So to complete our proof, we need to show that the outcome is uniquely determined by the function that constitutes the state, independent of the alias that happens to be used to name it. In providing this justification, we will essentially be verifying that the state set we have selected incorporates enough information about the bi-directional scanning done by A to allow A' to accomplish the same recognition with only a unidirectional scan!

Claim: for each $x,y \in \Sigma^*$, if $\delta_x = \delta_y$, then for all $\lambda \in \Sigma$, $\delta_{x\lambda} = \delta_{y\lambda}$.

Proof of claim:

case 1: assume either x or y is ε, say $\delta_\varepsilon = \delta_y$

We show in this case that y=ε by contradiction. Suppose $y = y'\lambda$ where $\lambda \in \Sigma$. Now $\delta_\varepsilon = \delta_y$ so $\delta_y(\theta) = \delta_\varepsilon(\theta) = s_0$ as noted above. Therefore $s_0 y'\lambda \vdash^* y'\lambda s_0$. But now consider the last step of this run, say $s_0 y'\lambda \vdash^* y't\lambda \vdash y'\lambda s_0$ for some state t. But then $\delta_y(t) = s_0$, while $\delta_\varepsilon(t) = \theta$. But this contradicts $\delta_\varepsilon = \delta_y$ and hence we must have $y = \varepsilon = x$ and so $\delta_{x\lambda} = \delta_\lambda = \delta_{y\lambda}$ and the claim is proven for case 1.

case 2: assume that $x \neq \varepsilon \neq y$ and $\delta_x = \delta_y$

subcase A: consider $\delta_{x\lambda}(s)$ and $\delta_{y\lambda}(s)$ where $s \in S$

subcase i: $\delta(s, \lambda) = (t, \mathcal{S})$

Then $\delta_{x\lambda}(s) = \delta_{y\lambda}(s) = t$

subcase ii: $\delta(s, \lambda) = (t, /)$

Since neither x nor y are null, we have that $x = x'\lambda_1$ and $y = y'\lambda_2$ for some $\lambda_1, \lambda_2 \in \Sigma$. Then consider the two runs $xs\lambda \vdash x't\lambda_1\lambda$ and $ys\lambda \vdash y't\lambda_2\lambda$. Since $\delta_x = d_y$, either

subcase a: $\delta_x(t) = \delta_y(t) = \theta$

Then these runs fail to move off the right of $x'\lambda_1$ and $y'\lambda_2$, respectively. Therefore the same is true of the runs starting from $xs\lambda$ and $ys\lambda$ and so $\delta_{x\lambda}(s) = \delta_{y\lambda}(s) = \theta$.

or

subcase b: there is $s_1 \in S$ with $\delta_x(t) = \delta_y(t) = s_1$

Thus in this case the two runs continue, respectively, as $xs\lambda \vdash\!\!- x't\lambda_1\lambda \vdash\!\!-^{*} xs_1\lambda$, and $ys\lambda \vdash\!\!- y't\lambda_2\lambda \vdash\!\!-^{*} y_1s_1\lambda$. Therefore $\delta_{x\lambda}(s) = \delta_{x\lambda}(s_1)$ and $\delta_{y\lambda}(s) = \delta_{y\lambda}(s_1)$. Now we again apply subcase A analysis to s_1, and again will be led to options (i) or (ii). If option (i) applies, then $\delta_{x\lambda}(s_1) = \delta_{y\lambda}(s_1)$ and hence $\delta_{x\lambda}(s) = \delta_{y\lambda}(s)$ and subcase A is concluded. Also by option (iia) we reach the same conclusion. By option (iib) we obtain still another state, say s_2, so that $\delta_{x\lambda}(s) = \delta_{x\lambda}(s_1) = \delta_{x\lambda}(s_2)$, and $\delta_{y\lambda}(s) = \delta_{y\lambda}(s_1) = \delta_{y\lambda}(s_2)$. Continued repetition of this analysis eventually either leads to the conclusion that the two results in subcase A are the same, or to an unending series of states s, s_1, s_2, s_3, … for which the analysis must be repeated. But in this latter circumstance, since there are finitely many states, there must at some point be a duplication of state, say $s_m = s_n$. But once a repetition occurs, it must be repeated indefinitely, and this shows that both runs enter an infinite loop and so $\delta_{x\lambda}(s) = \theta = \delta_{y\lambda}(s)$.

Therefore subcase A is complete.

subcase B: consider $\delta_{x\lambda}(\theta)$ and $\delta_{y\lambda}(\theta)$ [and case 2 assumptions still apply]

subcase i: $\delta_x(\theta) = \delta_y(\theta) = \theta$

In this case neither of the runs starting s_0x nor s_0y will move off the right of the input. But then neither will the runs $s_0x\lambda$ or $s_0y\lambda$, and therefore $\delta_{x\lambda}(\theta) = \delta_{y\lambda}(\theta) = \theta$.

subcase ii: $\delta_x(\theta) = \delta_y(\theta) = s$

Then $s_0x\lambda \vdash\!\!-^{*} xs\lambda$ and $s_0y\lambda \vdash\!\!-^{*} ys\lambda$, and so $\delta_{x\lambda}(\theta) = \delta_{y\lambda}(s)$ and $\delta_{y\lambda}(\theta) = \delta_{y\lambda}(s)$, and this immediately reduces to subcase A.

Thus we have shown that for every possible argument to $\delta_{x\lambda}$ and $\delta_{y\lambda}$, we obtain the same result provided that $\delta_x = \delta_y$ and so $\delta_{x\lambda} = \delta_{y\lambda}$ This completes the proof of our claim, and the proof of Theorem 1.4.1.

❑

The construction in this theorem is an excellent illustration of time/space trade-off possibilities in computing. Our construction here shows that an N state 2W-DFA which may require a significant amount of additional time doing its bi-directional scan of the input can always be replaced by a device using a single pass over the input. However, to achieve this one-pass time efficiency, the DFA (at least in our construction) may be required to have $(N+1)^{N+1}$ states! This latter is the worst-case bound determined by noting that the states in our DFA are functions with N+1 arguments, each of which could be associated with one of N+1 possible results.

Example 1.4.3.

We illustrate the construction in the proof of Theorem 1.4.1 by applying it to the 2W-DFA of Example 1.4.1. The first step is to develop the state set for the DFA. As we have noted a few times previously, we do not necessarily need all the $4^4 = 256$ states predicted by this construction. We only need those states reachable from the start state δ_ε, and the construction proceeds state-by-state from this starting point. When all reachable states have been constructed, the process will be complete. The details are quite tedious, and it may be help to look ahead from time to time at the completed DFA which is depicted at the end of the example.

Each of the functions δ_x will be a function on the set {a, b, c, θ}. We will write these functions as 2×4 arrays whose top row lists the arguments, and whose bottom row lists the result the function yields for the corresponding argument. For instance, in the proof of Theorem 1.4.1 it was observed that for the null string we have $\delta_\varepsilon(a) = \delta_\varepsilon(b) = \delta_\varepsilon(c) = \theta$, and $\delta_\varepsilon(\theta) = a$. So for function δ_ε we will write $\delta_\varepsilon = \begin{bmatrix} a & b & c & \theta \\ \theta & \theta & \theta & a \end{bmatrix}$.

Now from the run analysis

a0 |— 0a,
b0 |— 0c,
c0 |— 0c,

we have that $\delta_0 = \begin{bmatrix} a & b & c & \theta \\ a & c & c & a \end{bmatrix}$. And from

a1 |— off left,
b1 |— off left,

c1 |— 1a,

we have that $\delta_1 = \begin{bmatrix} a & b & c & \theta \\ \theta & \theta & a & \theta \end{bmatrix}$.

Then for δ_{00} we investigate the runs

0a0 |— 00a
0b0 |— 00c,
0c0 |— 00c,
a00 |— 0a0 |— 00a,

and therefore $\delta_{00} = \delta_0$.

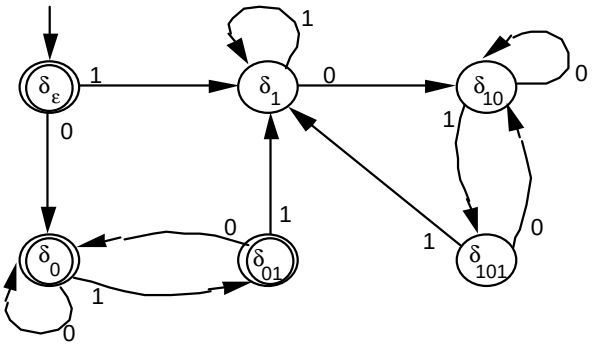
By similar analysis we find that $\delta_{11} = \delta_1$, and

$\delta_{01} = \begin{bmatrix} a & b & c & \theta \\ a & a & a & a \end{bmatrix}$, and $\delta_{10} = \begin{bmatrix} a & b & c & \theta \\ a & c & c & \theta \end{bmatrix}$.

Some effort on the direct computation of the functions can be avoided by noting equivalencies that have already been determined. For instance, in the DFA $\delta_{000} = \delta'(\delta_\varepsilon,000) = \delta'(\delta_{00}, 0) = \delta'(\delta_0, 0) = \delta_{00} = \delta_0$. Hence $\delta_{000} = \delta_{00} = \delta_0$, and $\delta_{001} = \delta_{01}$. We illustrate the determination of the δ_x functions for one more case. With x=011 we have the runs

01a1 |— 0b11 |— b011 |— 0c11 |— 01a1 |— 0b11 |— … infinite loop,
01b1 |— 0b11 |— … infinite loop,
01c1 |— 011a,
a011 |— 0a11 |— b011 |— … infinite loop,

and therefore $\delta_{011} = \delta_1$. Similarly we find that $\delta_{010} = \delta_0$, $\delta_{100} = \delta_{10}$, $\delta_{110} = \delta_{10}$, $\delta_{111} = \delta_1$, and $\delta_{101} = \begin{bmatrix} a & b & c & \theta \\ a & a & a & \theta \end{bmatrix}$. Finally, it can be determined that $\delta_{1010} = \delta_{10}$ and $\delta_{1011} = \delta_1$. A summary of the distinct (reachable) functions, and their transitions, is shown in the DFA state diagram below.



Notice that the language recognized by this DFA is the same as that noted for the 2W-DFA we started from, but is accomplished with a unidirectional scan.
❑

Since Theorem 1.4.1 identifies the 2W-DFA recognized languages with the regular languages, we have no need for separate investigation of their properties. It might be asked if anything is gained by providing for non-determinism or ε-moves in the two-way case. It turns out that these features fail to increase recognition power in this case as well. Other computational behaviors have also been studied. For instance, providing one-way scanning plus the ability to "rewind" the input to the beginning, and still exactly the regular sets are recognized. The interested reader can consult [Per 90] for broader developments.

Section 1.5: Summary

The regular expressions provide an initial, machine independent means of precisely describing collections of sequences — that is, *languages*. The general construction of strings in a collection is reflected in its regular expression, and a number of properties of the languages that are so described can be readily determined by an examination of this description. However, some properties are not so evident from such a description, including such basic processing problems as deciding if a candidate string belongs to the intended collection or not. The idea of a mechanical processing agent (DFA) that places each string in one of two categories is next introduced. Despite its

orientation to recognition rather than generation, the DFA (plus several significant variations) are found to be equivalent to the regular expression mechanism. In terms of understanding the limits of this basic model of computation, in this chapter we have taken the positive view and explored alternative means of expressing what *is* possible.

The availability of contrasting mechanisms permits the determination of many properties of the resulting family of regular languages. In addition to aiding the discovery of these properties, this development of contrasting but equivalent mechanisms establishes a pattern of analysis that will be continued, and extended, throughout the remainder of this book. Finally, while we have eschewed the development of explicit algorithms to resolve various processing problems, it is important to recognize that the basis for such algorithms has been established. The existence (or non-existence) of algorithms that can resolve various questions of interest is another theme that will receive increasing consideration as our investigations unfold.

Exercises.

1.1. Show that language concatenation is an associative operation.

1.2. Show that language concatenation distributes over set union (from either side).

1.3. Provide a regular expression and justify its correctness for each of the following languages over $\Sigma = \{0, 1\}$:
(a) all sequences with the same second and next-to-last letter
(b) all sequences whose first and last letter are different
(c) all sequences of even length
(d) all sequences with three or more '1's
(e)$^{\square}$ all sequences with an odd number of '1's
(f) all sequences with no two successive symbols the same
(g) all sequences with no two successive '0's
(h)* all sequences of even length with an odd number of '1's
(i) all sequences of two or more characters whose next-to-last character is 0
(j) all sequences *not* in $\mu((0^*1)^*)$

(k)¤ all sequences that do not contain 01 as a subsequence

1.4¤. For $\Sigma=\{1,2\}$, provide a regular expression that describes the strings $x = \lambda_1\lambda_2 \ldots \lambda_k$, ($\lambda_i \in \Sigma, 1 \le i \le k$) so that $\lambda_1+\lambda_2+ \ldots +\lambda_k$ is a multiple of 3 (λ_i is regarded as an integer in this summation, not a letter). Fully justify your answer.

1.5. Prove each of the following parts of Theorem 1.1.1
(a) part xi
(b) part xii
(c) part xiv

1.6. How do we know that the positive closure($^+$) of a regular language is regular?

1.7. For $\Sigma = \{a,b\}$ prove or disprove $\mu(a(a+b)^*) = \mu((ab^*)^*ab^*)$.

1.8. Prove or disprove whether the following are valid identities for all regular expressions α and β
(a) $(\alpha^* \beta^*)^* \equiv (\alpha^* \beta)^*$
(b) $(\alpha^* \beta)^* \equiv \varepsilon + (\alpha+\beta)^* \beta$

1.9. Prove Lemma 1.1.2.

1.10. Show that if α is a regular expression with the property that $\varepsilon \in \mu(\alpha)$, and whenever β_1 and β_2 are regular expressions so that $\mu(\beta_1) \subseteq \mu(\alpha)$ and $\mu(\beta_2) \subseteq \mu(\alpha)$, then $\mu(\beta_1 \bullet \beta_2) \subseteq \mu(\alpha)$, then there is a regular expression γ so that $\alpha \equiv \gamma^*$.

1.11. Describe and justify a series of analysis steps (i.e., an algorithm) that given any regular expression α, determines whether or not $\varepsilon \in \mu(\alpha)$.

1.12. The **composition** of two substitutions, σ_1 and σ_2, $\sigma_1 \circ \sigma_2$ is defined by $[\sigma_1 \circ \sigma_2](\lambda) = \sigma_2(\sigma_1(\lambda))$ for each $\lambda \in \Sigma$, and this clearly determines

another substitution. Show that the composition of two regular substitutions is another regular substitution.

1.13. For substitutions σ, σ_1, and σ_2 we can define new substitutions as follows:

- the **union** of two substitutions $[\sigma_1 \cup \sigma_2](\lambda) = \sigma_1(\lambda) \cup \sigma_2(\lambda)$ for each $\lambda \in \Sigma$,
- the **concatenation** of two substitutions $[\sigma_1 \bullet \sigma_2](\lambda) = \sigma_1(\lambda) \bullet \sigma_2(\lambda)$ for each $\lambda \in \Sigma$,
- the **star** of a substitution $[\sigma^*](\lambda) = (\sigma(\lambda))^*$ for each $\lambda \in \Sigma$.

Show that every regular substitution can be obtained from finite substitutions (i.e., those such that $\sigma(\lambda)$ is a finite language for each $\lambda \in \Sigma$) by finitely many applications of substitution union, concatenation, and star.

1.14. Prove that the language of Example 1.2.8 is $(1+01)^* (\varepsilon+0)$.

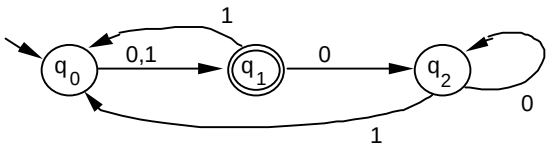
1.15. Provide an acceptor for each of the languages in Exercise 1.3 and justify your answers.

1.16. Construct a DFA over $\Sigma = \{0,1\}$ which accepts exactly those sequences which end with 00. Provide convincing justification that your DFA is correct.

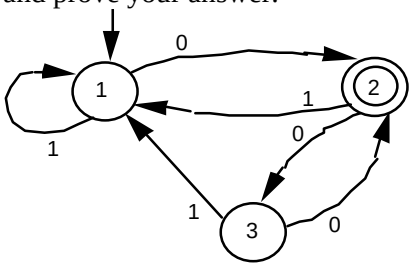
1.17. Provide a DFA that recognizes the complement of 0^*1^* and justify its correctness.

1.18. Provide a DFA over $\Sigma=\{0,1\}$ which accepts exactly those strings that have both 01 *and* 10 as (possibly overlapping) substrings, in either order. Give convincing justification that your solution accepts all such strings and no others.

1.19. Determine whether or not the regular expression $((0+1)0^*1)^* (0+1)$ describes exactly the language accepted by the DFA below and justify your answer.



1.20. Determine whether or not the regular expression $0\ (00)^* + (1+01+001)^*\ 0$ and the DFA in the state diagram below are equivalent, and prove your answer.



1.21. Prove that for a DFA $\lambda \in \Sigma$, $x,y \in \Sigma^*$, and $s \in S$,

(a) $\delta^*(s, \lambda x) = \delta^*(\delta(s,\lambda), x)$,

(b) $\delta^*(s, xy) = \delta^*(\delta^*(s,x), y)$.

1.22. What is the smallest DFA that recognizes $\mu(0^*+1^*)$?

1.23. Construct a regular expression that defines the same language as the acceptor given below (start =1, final={2,3}). Justify your answer.

δ	a	b
1	1	2
2	3	3
3	3	1

1.24. Show whether or not the regular expression 0(01)*0(0+01)* and the DFA with start state =1 and final states = {6} defined by the transition function below describe the same language.

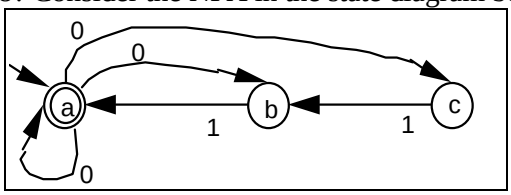
δ	0	1
1	2	7
2	3	7
3	4	2
4	5	6
5	7	6
6	7	6
7	7	7

1.25. Show that if a DFA accepts any strings at all, then it accepts some string whose length is less than the number of states.

1.26. Show that for regular languages R_1 and R_2, there is a single transition system that by choice of the accepting states yields a DFA recognizing each of the languages: R_1, R_2, $R_1 \cup R_2$, $R_1 \cap R_2$, $\neg R_1$, and $\neg R_2$.

1.27. Let $\Sigma = \{0, 1\}$ and define a DFA A over Σ with $L(A) = \{0101\}$ (i.e., A accepts just this one string). Is there a simpler NFA accepting this language? Justify your answers.

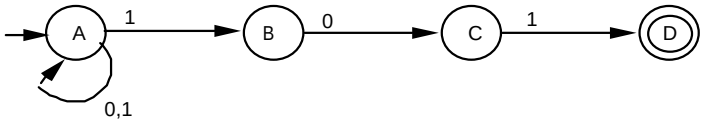
1.28¤. Consider the NFA in the state diagram below.



(a) Give a string beginning with '0' that is *not* accepted.

(b) Produce an equivalent DFA using the power set construction showing start and final states, and clearly indicating which subset of {a,b,c} is represented by each state. Omit states that are unreachable from the start state.

1.29. Use the powerset construction (see proof of Theorem 1.2.2) to obtain a DFA that accepts the same language as the NFA given below. You need only exhibit the states which are reachable from the start state (rather than all $2^4 = 16$ states).



1.30¤ With Σ = {0,1}, construct an acceptor (DFA, NFA or ε-NFA) that recognizes exactly those strings that have either 101 or 0110 (or both) as substrings (i.e., L = $(0+1)^*$ (101+0110) $(0+1)^*$). Clearly explain the correct operation of your acceptor.

1.31. Provide an ε-NFA that accepts the language $(01+001+010)^*$. Give convincing justification that your solution accepts all such strings and no others.

1.32. Provide a DFA for the language of Exercise 1.32 and give convincing justification that your solution accepts all such strings and no others.

1.33. Construct ε-NFAs recognizing each of the languages in Example 1.1.3.

1.34. Construct a DFA equivalent to the NFA in Example 1.2.4.

1.35. Show that for an ε-NFA, ε-closure(S∪T) = ε-closeure(S) ∪ ε-closure(T).

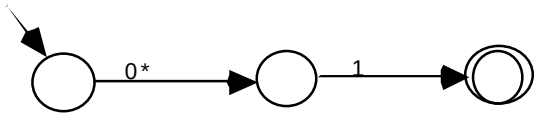
1.36. Show that for an ε-NFA, ε-closure(ε-closure(T)) = ε-closure(T) for each subset T of states.

1.37. Provide a proof of Lemma 1.2.3.

1.38. For Σ={0,1}, consider the set of all sequences of Σ^* which have an odd number of '0's and which do not end with '00'. Devise both a regular expression and an ε-NFA (or NFA or DFA) which accept this language, and show that they are both correct.

1.39. We define a new variety of acceptor in this problem, called a **meta-NFA**. A meta-NFA is just like an ε-NFA except that the transitions in a meta-NFA over alphabet Σ are described by *regular expressions* rather

than single letters. Since either ε, ∅ or any single letter of Σ is a regular expression, meta-NFA can be regarded as generalizations of ε-NFA. In a run of a given input sequence, state transitions are associated with reading an initial segment of the remaining input that is described by the regular expression for that transition. For example, depicted below is a simple meta-NFA that accepts 0^*1 (and has no cycles). Clearly every ε-NFA *is* a meta-NFA, but are meta-NFA equivalent to ε-NFA?



1.40. Prove that if a state of a transition system is reachable from another, then it is reachable using an input sequence of length N-1 or less, where the transition system has N states.

1.41. Show that if A is a strongly connected DFA, then either L(A) is empty or L(A) is infinite.

1.42. A **congruence** on a DFA A is an equivalence relation ρ on the state set such that for any two states s, s', if s ρ s', then δ(s, λ) ρ δ(s', λ) for all λ∈Σ. Show that if A is a DFA with no non-trivial congruences (there are two trivial congruences — one with all states in one class, and the other with each state in a separate class), then either
(i) A has two states, or
(ii) A has at most one state that is not a generator.

1.43. A DFA is a **permutation machine** if for each λ∈Σ, the next-state mapping is a permutation of the states. Show that if A is a permutation machine that has a generator state, then A is strongly connected.

1.44. A DFA is **nilpotent** if there exists a natural number N and a state r so that for all $x \in \Sigma^*$ with len(x) ≥ N, and all states s, δ(s,x) = r. Show that $L \subseteq \Sigma^*$ is either finite or co-finite (i.e., its complement is finite) if and only if L is accepted by a nilpotent DFA.

1.45. Provide an equivalent regular expression, and justification of its equivalence, for the acceptor in
(a) Example 1.3.1
(b) Example 1.2.2
(c) Example 1.2.3 (A_1)
(d) Example 1.2.3 (A_2)
(e) Example 1.2.3 (A_3)
(f) Example 1.2.4
(g) Example 1.2.5
(h) Example 1.2.8 (A)

1.46. Show that for each $X,Y \subseteq \Sigma^*$ the reversal operation has the following properties
(a) $(X \cup Y)^R = X^R \cup Y^R$
(b) $(X \bullet Y)^R = Y^R \bullet X^R$
(c) $(X^*)^R = (X^R)^*$

1.47. Show that if $L \subseteq \Sigma^*$ is regular, and $\lambda \in \Sigma$, then the language obtained from L by removing all occurrences of λ from strings of L is also regular.

1.48. For $h: \Sigma^* \rightarrow \Sigma^*$ define $Fixed(h) = \{x \in \Sigma^* \mid h(x) = x\}$. Show that if h is a homomorphism, there is a finite set F so that $F^* = Fixed(h)$.

1.49*. A (single) *infinite* sequence $\lambda_1\lambda_2 \ldots \lambda_n \ldots$ with λ_i from a (finite) alphabet Σ is **regular** if there are only finitely many (their number is called the **index**) distinct infinite subsequences among all its postfixes, $\lambda_k\lambda_{k+1}\lambda_{k+2} \ldots$ ($k \geq 1$). So for example (with $\Sigma=\{a,b\}$), the infinite sequence $\lambda_1\lambda_2\lambda_3 \ldots$, where

- $\lambda_i=a$ for all i (i.e., a^∞ = aa … a …), is regular with index 1;
- $\lambda_i=a$ for i odd, and $\lambda_i=b$ for i even (i.e., $(ab)^\infty$ = abab … ab …), is regular with index 2; and
- $\lambda_i=b$ if $i=n+n*(n+1)/2$ for some $n \geq 1$, and $\lambda_i=a$ otherwise (i.e., abaabaaabaaaab …), is not regular (why not?).

Show that for a regular infinite sequence, $\sigma = \lambda_1\lambda_2 \ldots \lambda_n \ldots$, the (infinite) language consisting of all its finite *prefixes*, $L_\sigma = \{\lambda_1\lambda_2 \ldots \lambda_k \mid k \geq 0\}$, is a regular set of (finite) sequences.

1.50* For 2W-DFA A, define loop(A) = $\{x \in \Sigma^* \mid s_0x$ initiates an infinite run$\}$. Is loop(A) necessarily regular? Justify your answer.

1.51* Consider the family of alphabets $\Sigma_n = \{¢, a_1, a_2, \ldots, a_n\}$ with n+1 symbols (n≥1). Define the languages $L_n = \{¢x \mid x \in (\Sigma_n - ¢)^*$ and x has at least one occurrence of each a_k, $1 \leq k \leq n\}$, one over each alphabet Σ_n (n≥1). Devise a 2n+2 state 2-way DFA to accept L_n. How many states do you require for an ordinary DFA? Justify your answers, and contrast the difference for, say, n=10.