FCS Week 10 Reading Note

Notebook:	Fundamentals of Computer Science		
Created:	2021-04-13 10:30 AM	**Updated:**	2021-05-02 7:57 PM
Author:	SUKHJIT MANN		

Cornell Notes	**Topic:** Regular Languages: Part 2	Course: BSc Computer Science Class: CM1025 Fundamentals of Computer Science[Reading] Date: May 02, 2021

Essential Question:

What is a regular expression and/or language?

Questions/Cues:

- What is a regular language?
- What are regular operations?
- What are closure properties?
- What are regular expressions?
- What is the formal definition of a regular expression?
- What is a GNFA?
- What is a non-regular language and the pumping lemma?

FORMAL DEFINITION OF COMPUTATION

So far we have described finite automata informally, using state diagrams, and with a formal definition, as a 5-tuple. The informal description is easier to grasp at first, but the formal definition is useful for making the notion precise, resolving any ambiguities that may have occurred in the informal description. Next we do the same for a finite automaton's computation. We already have an informal idea of the way it computes, and we now formalize it mathematically.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M ***accepts*** w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,

2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$, and

3. $r_n \in F$.

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state. We say that M ***recognizes language*** A if $A = \{w |\, M \text{ accepts } w\}$.

DEFINITION 1.16

A language is called a ***regular language*** if some finite automaton recognizes it.

EXAMPLE 1.17

Take machine M_5 from Example 1.13. Let w be the string

$$\texttt{10}\langle\text{RESET}\rangle\texttt{22}\langle\text{RESET}\rangle\texttt{012}.$$

Then M_5 accepts w according to the formal definition of computation because the sequence of states it enters when computing on w is

$$q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0,$$

which satisfies the three conditions. The language of M_5 is

$$L(M_5) = \{w |\, \text{the sum of the symbols in } w \text{ is } 0 \text{ modulo } 3, \\ \text{except that } \langle\text{RESET}\rangle \text{ resets the count to } 0\}.$$

As M_5 recognizes this language, it is a regular language. ■

THE REGULAR OPERATIONS

In the preceding two sections, we introduced and defined finite automata and regular languages. We now begin to investigate their properties. Doing so will help develop a toolbox of techniques for designing automata to recognize particular languages. The toolbox also will include ways of proving that certain other languages are nonregular (i.e., beyond the capability of finite automata).

In arithmetic, the basic objects are numbers and the tools are operations for manipulating them, such as + and ×. In the theory of computation, the objects are languages and the tools include operations specifically designed for manipulating them. We define three operations on languages, called the ***regular operations***, and use them to study properties of the regular languages.

DEFINITION 1.23

Let A and B be languages. We define the regular operations ***union***, ***concatenation***, and ***star*** as follows:

- **Union**: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- **Concatenation**: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$.
- **Star**: $A^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$.

You are already familiar with the union operation. It simply takes all the strings in both A and B and lumps them together into one language.

The concatenation operation is a little trickier. It attaches a string from A in front of a string from B in all possible ways to get the strings in the new language.

The star operation is a bit different from the other two because it applies to a single language rather than to two different languages. That is, the star operation is a ***unary operation*** instead of a ***binary operation***. It works by attaching any number of strings in A together to get a string in the new language. Because "any number" includes 0 as a possibility, the empty string ε is always a member of A^*, no matter what A is.

EXAMPLE 1.24

Let the alphabet Σ be the standard 26 letters $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. If $A = \{\mathtt{good}, \mathtt{bad}\}$ and $B = \{\mathtt{boy}, \mathtt{girl}\}$, then

$$A \cup B = \{\mathtt{good}, \mathtt{bad}, \mathtt{boy}, \mathtt{girl}\},$$

$$A \circ B = \{\mathtt{goodboy}, \mathtt{goodgirl}, \mathtt{badboy}, \mathtt{badgirl}\}, \text{ and}$$

$$A^* = \{\varepsilon, \mathtt{good}, \mathtt{bad}, \mathtt{goodgood}, \mathtt{goodbad}, \mathtt{badgood}, \mathtt{badbad}, \\ \mathtt{goodgoodgood}, \mathtt{goodgoodbad}, \mathtt{goodbadgood}, \mathtt{goodbadbad}, \ldots\}.$$

■

Let $\mathcal{N} = \{1, 2, 3, \ldots\}$ be the set of natural numbers. When we say that $\mathcal{N}$ is *closed under multiplication*, we mean that for any x and y in $\mathcal{N}$, the product $x \times y$ also is in $\mathcal{N}$. In contrast, $\mathcal{N}$ is not closed under division, as 1 and 2 are in $\mathcal{N}$ but 1/2 is not. Generally speaking, a collection of objects is ***closed*** under some operation if applying that operation to members of the collection returns an object still in the collection. We show that the collection of regular languages is closed under all three of the regular operations. In Section 1.3, we show that these are useful tools for manipulating regular languages and understanding the power of finite automata. We begin with the union operation.

THEOREM 1.25

The class of regular languages is closed under the union operation.

In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

PROOF IDEA We have regular languages A_1 and A_2 and want to show that $A_1 \cup A_2$ also is regular. Because A_1 and A_2 are regular, we know that some finite automaton M_1 recognizes A_1 and some finite automaton M_2 recognizes A_2. To prove that $A_1 \cup A_2$ is regular, we demonstrate a finite automaton, call it M, that recognizes $A_1 \cup A_2$.

This is a proof by construction. We construct M from M_1 and M_2. Machine M must accept its input exactly when either M_1 or M_2 would accept it in order to recognize the union language. It works by *simulating* both M_1 and M_2 and accepting if either of the simulations accept.

How can we make machine M simulate M_1 and M_2? Perhaps it first simulates M_1 on the input and then simulates M_2 on the input. But we must be careful here! Once the symbols of the input have been read and used to simulate M_1, we can't "rewind the input tape" to try the simulation on M_2. We need another approach.

THEOREM 1.26

The class of regular languages is closed under the concatenation operation.

In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

To prove this theorem, let's try something along the lines of the proof of the union case. As before, we can start with finite automata M_1 and M_2 recognizing the regular languages A_1 and A_2. But now, instead of constructing automaton M to accept its input if either M_1 or M_2 accept, it must accept if its input can be broken into two pieces, where M_1 accepts the first piece and M_2 accepts the second piece. The problem is that M doesn't know where to break its input (i.e., where the first part ends and the second begins). To solve this problem, we introduce a new technique called nondeterminism.

THEOREM 1.45

The class of regular languages is closed under the union operation.

PROOF IDEA We have regular languages A_1 and A_2 and want to prove that $A_1 \cup A_2$ is regular. The idea is to take two NFAs, N_1 and N_2 for A_1 and A_2, and combine them into one new NFA, N.

Machine N must accept its input if either N_1 or N_2 accepts this input. The new machine has a new start state that branches to the start states of the old machines with ε arrows. In this way, the new machine nondeterministically guesses which of the two machines accepts the input. If one of them accepts the input, N will accept it, too.

We represent this construction in the following figure. On the left, we indicate the start and accept states of machines N_1 and N_2 with large circles and some additional states with small circles. On the right, we show how to combine N_1 and N_2 into N by adding additional transition arrows.

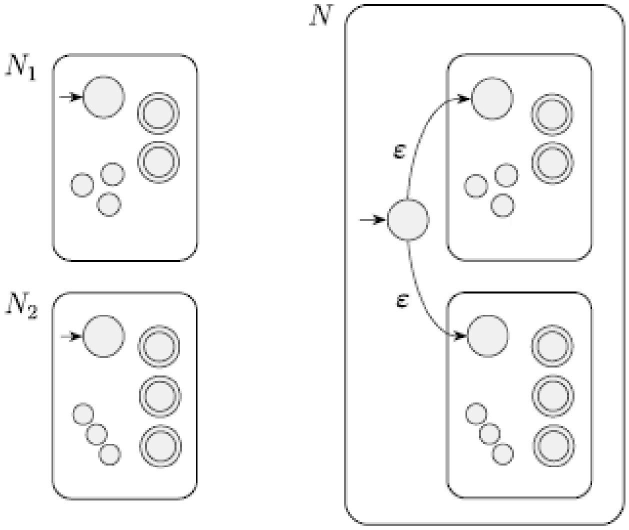


FIGURE 1.46

Construction of an NFA N to recognize $A_1 \cup A_2$

THEOREM 1.47

The class of regular languages is closed under the concatenation operation.

PROOF IDEA We have regular languages A_1 and A_2 and want to prove that $A_1 \circ A_2$ is regular. The idea is to take two NFAs, N_1 and N_2 for A_1 and A_2, and combine them into a new NFA N as we did for the case of union, but this time in a different way, as shown in Figure 1.48.

Assign N's start state to be the start state of N_1. The accept states of N_1 have additional ε arrows that nondeterministically allow branching to N_2 whenever N_1 is in an accept state, signifying that it has found an initial piece of the input that constitutes a string in A_1. The accept states of N are the accept states of N_2 only. Therefore, it accepts when the input can be split into two parts, the first accepted by N_1 and the second by N_2. We can think of N as nondeterministically guessing where to make the split.

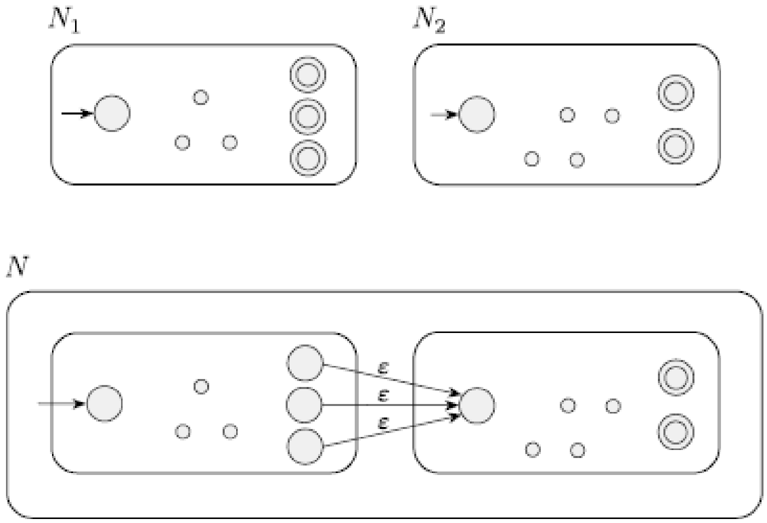


FIGURE 1.48
Construction of N to recognize $A_1 \circ A_2$

THEOREM 1.49

The class of regular languages is closed under the star operation.

PROOF IDEA We have a regular language A_1 and want to prove that A_1^* also is regular. We take an NFA N_1 for A_1 and modify it to recognize A_1^*, as shown in the following figure. The resulting NFA N will accept its input whenever it can be broken into several pieces and N_1 accepts each piece.

We can construct N like N_1 with additional ε arrows returning to the start state from the accept states. This way, when processing gets to the end of a piece that N_1 accepts, the machine N has the option of jumping back to the start state to try to read another piece that N_1 accepts. In addition, we must modify N so that it accepts ε, which always is a member of A_1^*. One (slightly bad) idea is simply to add the start state to the set of accept states. This approach certainly adds ε to the recognized language, but it may also add other, undesired strings. Exercise 1.15 asks for an example of the failure of this idea. The way to fix it is to add a new start state, which also is an accept state, and which has an ε arrow to the old start state. This solution has the desired effect of adding ε to the language without adding anything else.

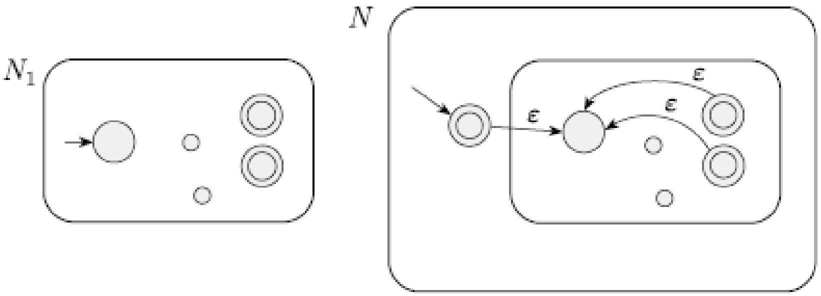


FIGURE 1.50
Construction of N to recognize A^*

1.3

REGULAR EXPRESSIONS

In arithmetic, we can use the operations + and × to build up expressions such as

$$(5+3)\times 4\,.$$

Similarly, we can use the regular operations to build up expressions describing languages, which are called ***regular expressions***. An example is:

$$(\mathtt{0}\cup\mathtt{1})\mathtt{0}^*.$$

The value of the arithmetic expression is the number 32. The value of a regular expression is a language. In this case, the value is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s. We get this result by dissecting the expression into its parts. First, the symbols 0 and 1 are shorthand for the sets $\{\mathtt{0}\}$ and $\{\mathtt{1}\}$. So $(\mathtt{0}\cup\mathtt{1})$ means $(\{\mathtt{0}\}\cup\{\mathtt{1}\})$. The value of this part is the language $\{\mathtt{0,1}\}$. The part $\mathtt{0}^*$ means $\{\mathtt{0}\}^*$, and its value is the language consisting of all strings containing any number of 0s. Second, like the × symbol in algebra, the concatenation symbol $\circ$ often is implicit in regular expressions. Thus $(\mathtt{0}\cup\mathtt{1})\mathtt{0}^*$ actually is shorthand for $(\mathtt{0}\cup\mathtt{1})\circ\mathtt{0}^*$. The concatenation attaches the strings from the two parts to obtain the value of the entire expression.

EXAMPLE 1.51

Another example of a regular expression is

$$(\mathtt{0} \cup \mathtt{1})^*.$$

It starts with the language $(\mathtt{0} \cup \mathtt{1})$ and applies the $*$ operation. The value of this expression is the language consisting of all possible strings of 0s and 1s. If $\Sigma = \{\mathtt{0},\mathtt{1}\}$, we can write Σ as shorthand for the regular expression $(\mathtt{0} \cup \mathtt{1})$. More generally, if Σ is any alphabet, the regular expression Σ describes the language consisting of all strings of length 1 over this alphabet, and Σ^* describes the language consisting of all strings over that alphabet. Similarly, $\Sigma^*\mathtt{1}$ is the language that contains all strings that end in a 1. The language $(\mathtt{0}\Sigma^*) \cup (\Sigma^*\mathtt{1})$ consists of all strings that start with a 0 or end with a 1. ■

In arithmetic, we say that $\times$ has precedence over $+$ to mean that when there is a choice, we do the $\times$ operation first. Thus in $2+3\times 4$, the 3×4 is done before the addition. To have the addition done first, we must add parentheses to obtain $(2+3)\times 4$. In regular expressions, the star operation is done first, followed by concatenation, and finally union, unless parentheses change the usual order.

FORMAL DEFINITION OF A REGULAR EXPRESSION

DEFINITION 1.52

Say that R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

In items 1 and 2, the regular expressions a and ε represent the languages $\{a\}$ and $\{\varepsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

Don't confuse the regular expressions ε and $\emptyset$. The expression ε represents the language containing a single string—namely, the empty string—whereas $\emptyset$ represents the language that doesn't contain any strings.

Seemingly, we are in danger of defining the notion of a regular expression in terms of itself. If true, we would have a ***circular definition***, which would be invalid. However, R_1 and R_2 always are smaller than R. Thus we actually are defining regular expressions in terms of smaller regular expressions and thereby avoiding circularity. A definition of this type is called an ***inductive definition***.

Parentheses in an expression may be omitted. If they are, evaluation is done in the precedence order: star, then concatenation, then union.

For convenience, we let R^+ be shorthand for RR^*. In other words, whereas R^* has all strings that are 0 or more concatenations of strings from R, the language R^+ has all strings that are 1 or more concatenations of strings from R. So $R^+ \cup \varepsilon = R^*$. In addition, we let R^k be shorthand for the concatenation of k R's with each other.

When we want to distinguish between a regular expression R and the language that it describes, we write $L(R)$ to be the language of R.

EXAMPLE 1.53

In the following instances, we assume that the alphabet Σ is $\{0,1\}$.

1. $0^*10^* = \{w|\ w$ contains a single $1\}$.
2. $\Sigma^*1\Sigma^* = \{w|\ w$ has at least one $1\}$.
3. $\Sigma^*001\Sigma^* = \{w|\ w$ contains the string 001 as a substring$\}$.
4. $1^*(01^+)^* = \{w|$ every 0 in w is followed by at least one $1\}$.
5. $(\Sigma\Sigma)^* = \{w|\ w$ is a string of even length$\}$.[5]
6. $(\Sigma\Sigma\Sigma)^* = \{w|$ the length of w is a multiple of $3\}$.
7. $01 \cup 10 = \{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w|\ w$ starts and ends with the same symbol$\}$.
9. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.
   The expression $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$, so the concatenation operation adds either 0 or ε before every string in 1^*.
10. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
11. $1^*\emptyset = \emptyset$.
    Concatenating the empty set to any set yields the empty set.
12. $\emptyset^* = \{\varepsilon\}$.
    The star operation puts together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, giving only the empty string.

[5] The ***length*** of a string is the number of symbols that it contains.

If we let R be any regular expression, we have the following identities. They are good tests of whether you understand the definition.

$R \cup \emptyset = R$.
Adding the empty language to any other language will not change it.

$R \circ \varepsilon = R$.
Joining the empty string to any string will not change it.

However, exchanging $\emptyset$ and ε in the preceding identities may cause the equalities to fail.

$R \cup \varepsilon$ may not equal R.
For example, if $R = \texttt{0}$, then $L(R) = \{\texttt{0}\}$ but $L(R \cup \varepsilon) = \{\texttt{0}, \varepsilon\}$.

$R \circ \emptyset$ may not equal R.
For example, if $R = \texttt{0}$, then $L(R) = \{\texttt{0}\}$ but $L(R \circ \emptyset) = \emptyset$.

Regular expressions are useful tools in the design of compilers for programming languages. Elemental objects in a programming language, called ***tokens***, such as the variable names and constants, may be described with regular expressions. For example, a numerical constant that may include a fractional part and/or a sign may be described as a member of the language

$$(\texttt{+} \cup \texttt{-} \cup \varepsilon)\left(D^{+} \cup D^{+}\,\texttt{.}\,D^{*} \cup D^{*}\,\texttt{.}\,D^{+}\right)$$

where $D = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}\}$ is the alphabet of decimal digits. Examples of generated strings are: 72, 3.14159, +7., and -.01.

Once the syntax of a programming language has been described with a regular expression in terms of its tokens, automatic systems can generate the ***lexical analyzer***, the part of a compiler that initially processes the input program.

EQUIVALENCE WITH FINITE AUTOMATA

Regular expressions and finite automata are equivalent in their descriptive power. This fact is surprising because finite automata and regular expressions superficially appear to be rather different. However, any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa. Recall that a regular language is one that is recognized by some finite automaton.

THEOREM 1.54

A language is regular if and only if some regular expression describes it.

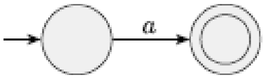
LEMMA 1.55

If a language is described by a regular expression, then it is regular.

PROOF IDEA Say that we have a regular expression R describing some language A. We show how to convert R into an NFA recognizing A. By Corollary 1.40, if an NFA recognizes A then A is regular.

PROOF Let's convert R into an NFA N. We consider the six cases in the formal definition of regular expressions.

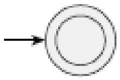
1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.



   Note that this machine fits the definition of an NFA but not that of a DFA because it has some states with no exiting arrow for each possible input symbol. Of course, we could have presented an equivalent DFA here; but an NFA is all we need for now, and it is easier to describe.

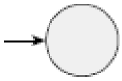
   Formally, $N = (\{q_1, q_2\},\ \Sigma,\ \delta,\ q_1,\ \{q_2\})$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.

2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$, and the following NFA recognizes $L(R)$.



   Formally, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, where $\delta(r, b) = \emptyset$ for any r and b.

3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$.



   Formally, $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, where $\delta(r, b) = \emptyset$ for any r and b.

4. $R = R_1 \cup R_2$.
5. $R = R_1 \circ R_2$.
6. $R = R_1^*$.

For the last three cases, we use the constructions given in the proofs that the class of regular languages is closed under the regular operations. In other words, we construct the NFA for R from the NFAs for R_1 and R_2 (or just R_1 in case 6) and the appropriate closure construction.

EXAMPLE 1.56

We convert the regular expression $(\mathtt{ab} \cup \mathtt{a})^*$ to an NFA in a sequence of stages. We build up from the smallest subexpressions to larger subexpressions until we have an NFA for the original expression, as shown in the following diagram. Note that this procedure generally doesn't give the NFA with the fewest states. In this example, the procedure gives an NFA with eight states, but the smallest equivalent NFA has only two states. Can you find it?

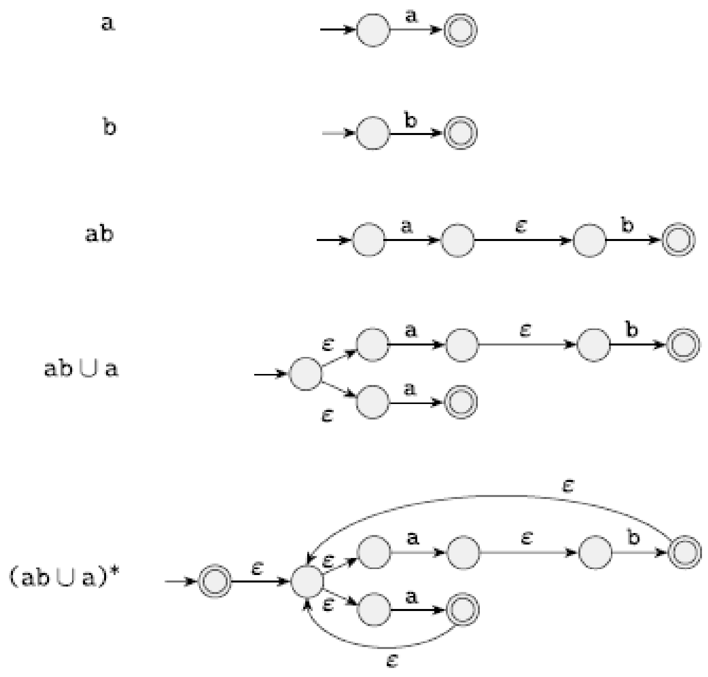


FIGURE 1.57
Building an NFA from the regular expression $(\mathtt{ab} \cup \mathtt{a})^*$

■

EXAMPLE 1.58

In Figure 1.59, we convert the regular expression $(\mathtt{a} \cup \mathtt{b})^*\mathtt{aba}$ to an NFA. A few of the minor steps are not shown.

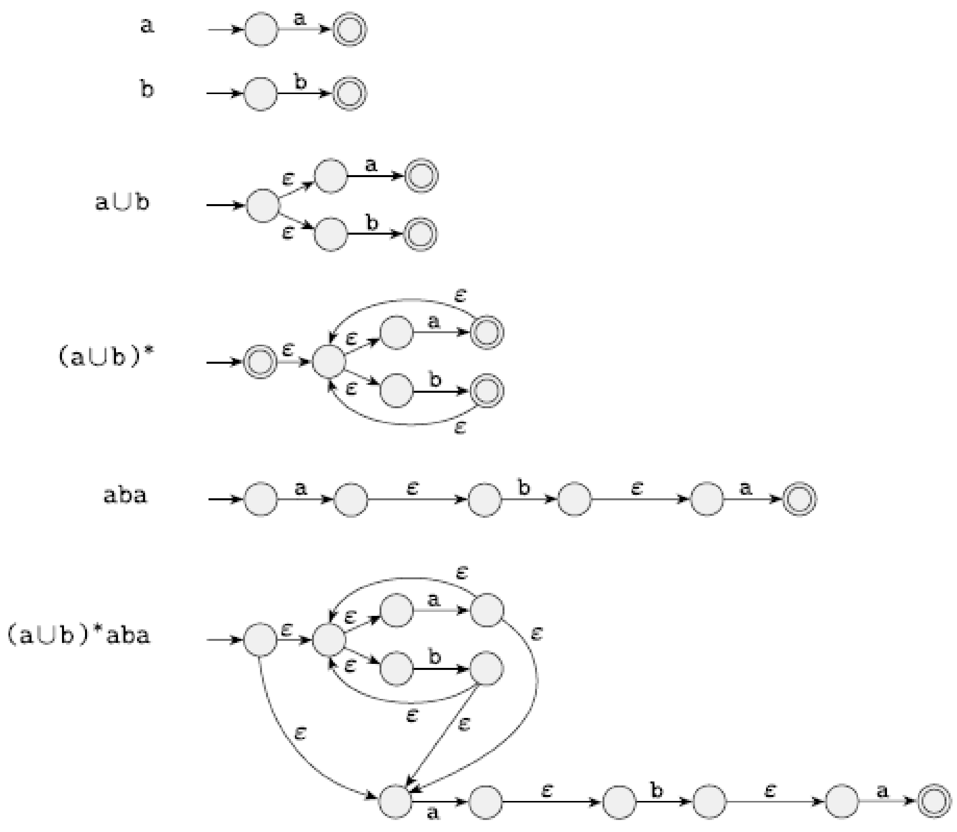


FIGURE 1.59
Building an NFA from the regular expression $(\mathtt{a} \cup \mathtt{b})^*\mathtt{aba}$

LEMMA 1.60

If a language is regular, then it is described by a regular expression.

PROOF IDEA We need to show that if a language A is regular, a regular expression describes it. Because A is regular, it is accepted by a DFA. We describe a procedure for converting DFAs into equivalent regular expressions.

We break this procedure into two parts, using a new type of finite automaton called a ***generalized nondeterministic finite automaton***, GNFA. First we show how to convert DFAs into GNFAs, and then GNFAs into regular expressions.

Generalized nondeterministic finite automata are simply nondeterministic finite automata wherein the transition arrows may have any regular expressions as labels, instead of only members of the alphabet or ε. The GNFA reads blocks of symbols from the input, not necessarily just one symbol at a time as in an ordinary NFA. The GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string described by the regular expression on that arrow. A GNFA is nondeterministic and so may have several different ways to process the same input string. It accepts its input if its processing can cause the GNFA to be in an accept state at the end of the input. The following figure presents an example of a GNFA.

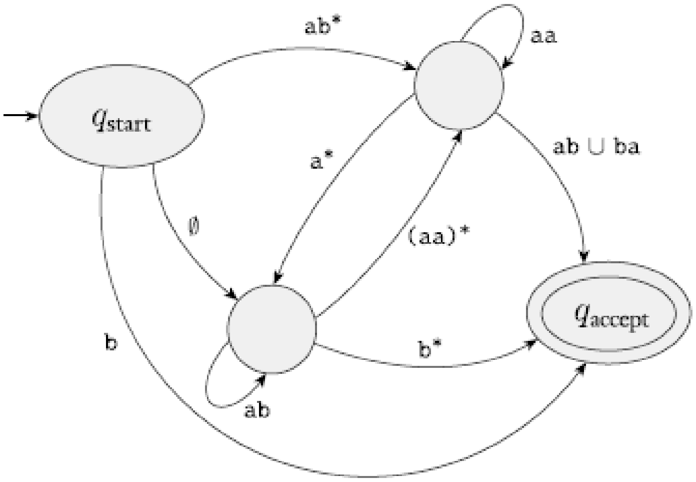


FIGURE 1.61
A generalized nondeterministic finite automaton

For convenience, we require that GNFAs always have a special form that meets the following conditions.

- The start state has transition arrows going to every other state but no arrows coming in from any other state.
- There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
- Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.

We can easily convert a DFA into a GNFA in the special form. We simply add a new start state with an ε arrow to the old start state and a new accept state with ε arrows from the old accept states. If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels. Finally, we add arrows labeled $\emptyset$ between states that had no arrows. This last step won't change the language recognized because a transition labeled with $\emptyset$ can never be used. From here on we assume that all GNFAs are in the special form.

Now we show how to convert a GNFA into a regular expression. Say that the GNFA has k states. Then, because a GNFA must have a start and an accept state and they must be different from each other, we know that $k \geq 2$. If $k > 2$, we construct an equivalent GNFA with $k - 1$ states. This step can be repeated on the new GNFA until it is reduced to two states. If $k = 2$, the GNFA has a single arrow that goes from the start state to the accept state. The label of this arrow is the equivalent regular expression. For example, the stages in converting a DFA with three states to an equivalent regular expression are shown in the following figure.

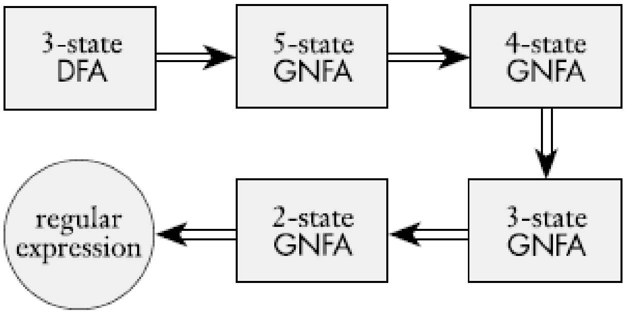


FIGURE 1.62
Typical stages in converting a DFA to a regular expression

The crucial step is constructing an equivalent GNFA with one fewer state when $k > 2$. We do so by selecting a state, ripping it out of the machine, and repairing the remainder so that the same language is still recognized. Any state will do, provided that it is not the start or accept state. We are guaranteed that such a state will exist because $k > 2$. Let's call the removed state q_{rip}.

After removing q_{rip} we repair the machine by altering the regular expressions that label each of the remaining arrows. The new labels compensate for the absence of q_{rip} by adding back the lost computations. The new label going from a state q_i to a state q_j is a regular expression that describes all strings that would take the machine from q_i to q_j either directly or via q_{rip}. We illustrate this approach in Figure 1.63.

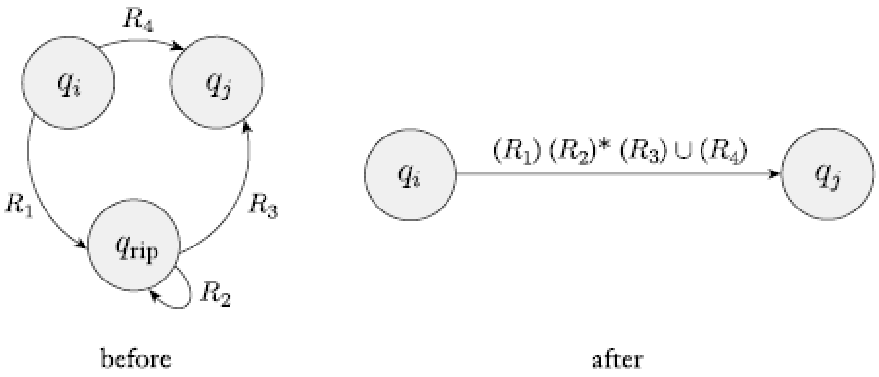


FIGURE 1.63
Constructing an equivalent GNFA with one fewer state

In the old machine, if

1. q_i goes to q_{rip} with an arrow labeled R_1,
2. q_{rip} goes to itself with an arrow labeled R_2,
3. q_{rip} goes to q_j with an arrow labeled R_3, and
4. q_i goes to q_j with an arrow labeled R_4,

then in the new machine, the arrow from q_i to q_j gets the label

$$(R_1)(R_2)^*(R_3) \cup (R_4).$$

We make this change for each arrow going from any state q_i to any state q_j, including the case where $q_i = q_j$. The new machine recognizes the original language.

PROOF Let's now carry out this idea formally. First, to facilitate the proof, we formally define the new type of automaton introduced. A GNFA is similar to a nondeterministic finite automaton except for the transition function, which has the form

$$\delta\colon \big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big) \longrightarrow \mathcal{R}.$$

The symbol $\mathcal{R}$ is the collection of all regular expressions over the alphabet Σ, and q_{start} and q_{accept} are the start and accept states. If $\delta(q_i, q_j) = R$, the arrow from state q_i to state q_j has the regular expression R as its label. The domain of the transition function is $\big(Q - \{q_{\text{accept}}\}\big) \times \big(Q - \{q_{\text{start}}\}\big)$ because an arrow connects every state to every other state, except that no arrows are coming from q_{accept} or going to q_{start}.

DEFINITION 1.64

A ***generalized nondeterministic finite automaton*** is a 5-tuple, $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where

1. Q is the finite set of states,
2. Σ is the input alphabet,
3. $\delta\colon (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \longrightarrow \mathcal{R}$ is the transition function,
4. q_{start} is the start state, and
5. q_{accept} is the accept state.

A GNFA accepts a string w in Σ^* if $w = w_1 w_2 \cdots w_k$, where each w_i is in Σ^* and a sequence of states $q_0, q_1, \ldots, q_k$ exists such that

1. $q_0 = q_{\text{start}}$ is the start state,
2. $q_k = q_{\text{accept}}$ is the accept state, and
3. for each i, we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$; in other words, R_i is the expression on the arrow from q_{i-1} to q_i.

Returning to the proof of Lemma 1.60, we let M be the DFA for language A. Then we convert M to a GNFA G by adding a new start state and a new accept state and additional transition arrows as necessary. We use the procedure CONVERT(G), which takes a GNFA and returns an equivalent regular expression. This procedure uses ***recursion***, which means that it calls itself. An infinite loop is avoided because the procedure calls itself only to process a GNFA that has one fewer state. The case where the GNFA has two states is handled without recursion.

```
CONVERT($G$):
1. Let $k$ be the number of states of $G$.
2. If $k = 2$, then $G$ must consist of a start state, an accept state, and a single
   arrow connecting them and labeled with a regular expression $R$.
   Return the expression $R$.
3. If $k > 2$, we select any state $q_{\text{rip}} \in Q$ different from $q_{\text{start}}$ and $q_{\text{accept}}$ and let
   $G'$ be the GNFA $(Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$, where
       $Q' = Q - \{q_{\text{rip}}\}$,
   and for any $q_i \in Q' - \{q_{\text{accept}}\}$ and any $q_j \in Q' - \{q_{\text{start}}\}$, let
       $\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4)$,
   for $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = \delta(q_{\text{rip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$.
4. Compute CONVERT($G'$) and return this value.
```

Next we prove that CONVERT returns a correct value.

CLAIM 1.65

For any GNFA G, CONVERT(G) is equivalent to G.

We prove this claim by induction on k, the number of states of the GNFA.

Basis: Prove the claim true for $k = 2$ states. If G has only two states, it can have only a single arrow, which goes from the start state to the accept state. The regular expression label on this arrow describes all the strings that allow G to get to the accept state. Hence this expression is equivalent to G.

Induction step: Assume that the claim is true for $k - 1$ states and use this assumption to prove that the claim is true for k states. First we show that G and G' recognize the same language. Suppose that G accepts an input w. Then in an accepting branch of the computation, G enters a sequence of states:

$$q_{\text{start}}, q_1, q_2, q_3, \ldots, q_{\text{accept}}.$$

If none of them is the removed state q_{rip}, clearly G' also accepts w. The reason is that each of the new regular expressions labeling the arrows of G' contains the old regular expression as part of a union.

If q_{rip} does appear, removing each run of consecutive q_{rip} states forms an accepting computation for G'. The states q_i and q_j bracketing a run have a new regular expression on the arrow between them that describes all strings taking q_i to q_j via q_{rip} on G. So G' accepts w.

Conversely, suppose that G' accepts an input w. As each arrow between any two states q_i and q_j in G' describes the collection of strings taking q_i to q_j in G, either directly or via q_{rip}, G must also accept w. Thus G and G' are equivalent.

The induction hypothesis states that when the algorithm calls itself recursively on input G', the result is a regular expression that is equivalent to G' because G' has $k - 1$ states. Hence this regular expression also is equivalent to G, and the algorithm is proved correct.

This concludes the proof of Claim 1.65, Lemma 1.60, and Theorem 1.54.

EXAMPLE 1.66

In this example, we use the preceding algorithm to convert a DFA into a regular expression. We begin with the two-state DFA in Figure 1.67(a).

In Figure 1.67(b), we make a four-state GNFA by adding a new start state and a new accept state, called s and a instead of q_{start} and q_{accept} so that we can draw them conveniently. To avoid cluttering up the figure, we do not draw the arrows labeled $\emptyset$, even though they are present. Note that we replace the label a, b on the self-loop at state 2 on the DFA with the label $\mathtt{a} \cup \mathtt{b}$ at the corresponding point on the GNFA. We do so because the DFA's label represents two transitions, one for a and the other for b, whereas the GNFA may have only a single transition going from 2 to itself.

In Figure 1.67(c), we remove state 2 and update the remaining arrow labels. In this case, the only label that changes is the one from 1 to a. In part (b) it was $\emptyset$, but in part (c) it is $\mathtt{b}(\mathtt{a} \cup \mathtt{b})^*$. We obtain this result by following step 3 of the CONVERT procedure. State q_i is state 1, state q_j is a, and q_{rip} is 2, so $R_1 = \mathtt{b}$, $R_2 = \mathtt{a} \cup \mathtt{b}$, $R_3 = \varepsilon$, and $R_4 = \emptyset$. Therefore, the new label on the arrow from 1 to a is $(\mathtt{b})(\mathtt{a} \cup \mathtt{b})^*(\varepsilon) \cup \emptyset$. We simplify this regular expression to $\mathtt{b}(\mathtt{a} \cup \mathtt{b})^*$.

In Figure 1.67(d), we remove state 1 from part (c) and follow the same procedure. Because only the start and accept states remain, the label on the arrow joining them is the regular expression that is equivalent to the original DFA.

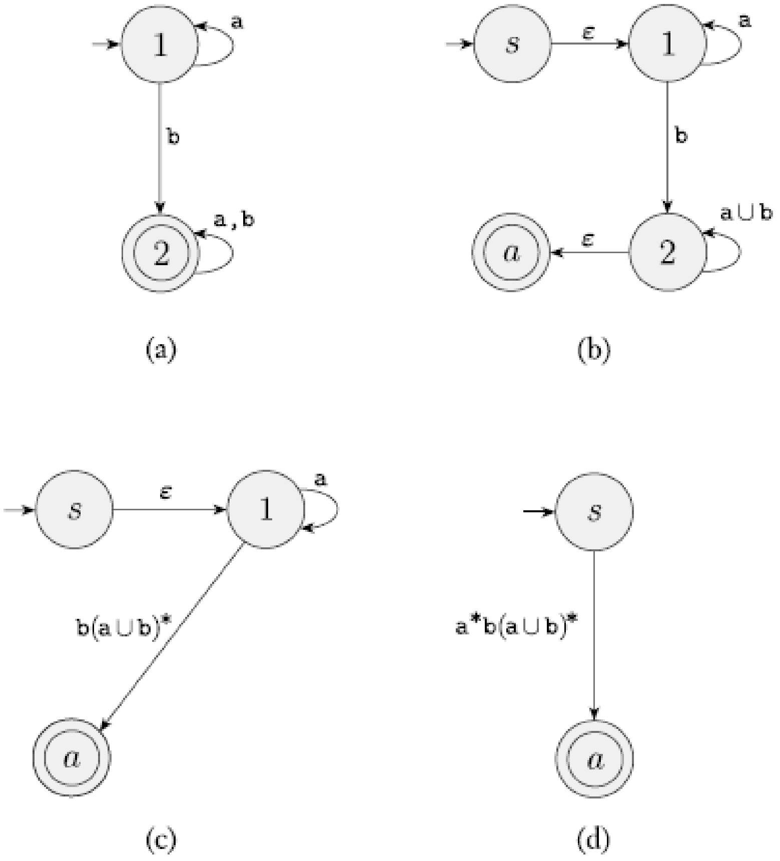
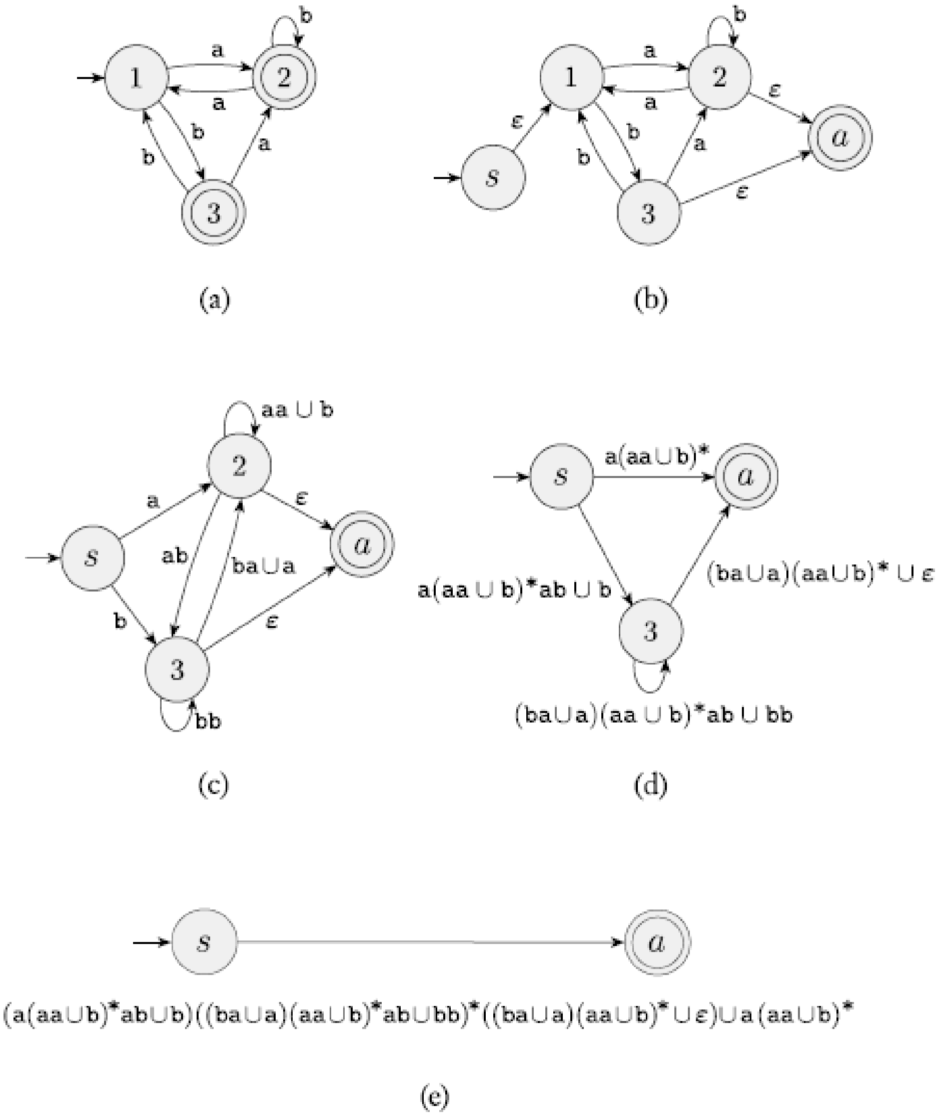


FIGURE 1.67
Converting a two-state DFA to an equivalent regular expression ■

EXAMPLE 1.68

In this example, we begin with a three-state DFA. The steps in the conversion are shown in the following figure.



(a) (b) (c) (d) (e)

FIGURE 1.69
Converting a three-state DFA to an equivalent regular expression ■

1.4

NONREGULAR LANGUAGES

To understand the power of finite automata, you must also understand their limitations. In this section, we show how to prove that certain languages cannot be recognized by any finite automaton.

Let's take the language $B = \{0^n1^n |\ n \geq 0\}$. If we attempt to find a DFA that recognizes B, we discover that the machine seems to need to remember how many 0s have been seen so far as it reads the input. Because the number of 0s isn't limited, the machine will have to keep track of an unlimited number of possibilities. But it cannot do so with any finite number of states.

Next, we present a method for proving that languages such as B are not regular. Doesn't the argument already given prove nonregularity because the number of 0s is unlimited? It does not. Just because the language appears to require unbounded memory doesn't mean that it is necessarily so. It does happen to be true for the language B; but other languages seem to require an unlimited number of possibilities, yet actually they are regular. For example, consider two languages over the alphabet $\Sigma = \{0,1\}$:

$C = \{w|\ w$ has an equal number of 0s and 1s$\}$, and

$D = \{w|\ w$ has an equal number of occurrences of 01 and 10 as substrings$\}$.

At first glance, a recognizing machine appears to need to count in each case, and therefore neither language appears to be regular. As expected, C is not regular, but surprisingly D is regular![6] Thus our intuition can sometimes lead us astray, which is why we need mathematical proofs for certainty. In this section, we show how to prove that certain languages are not regular.

THE PUMPING LEMMA FOR REGULAR LANGUAGES

Our technique for proving nonregularity stems from a theorem about regular languages, traditionally called the ***pumping lemma***. This theorem states that all regular languages have a special property. If we can show that a language does not have this property, we are guaranteed that it is not regular. The property states that all strings in the language can be "pumped" if they are at least as long as a certain special value, called the ***pumping length***. That means each such string contains a section that can be repeated any number of times with the resulting string remaining in the language.

THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Recall the notation where $|s|$ represents the length of string s, y^i means that i copies of y are concatenated together, and y^0 equals ε.

When s is divided into xyz, either x or z may be ε, but condition 2 says that $y \neq \varepsilon$. Observe that without condition 2 the theorem would be trivially true. Condition 3 states that the pieces x and y together have length at most p. It is an extra technical condition that we occasionally find useful when proving certain languages to be nonregular. See Example 1.74 for an application of condition 3.

PROOF IDEA Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA that recognizes A. We assign the pumping length p to be the number of states of M. We show that any string s in A of length at least p may be broken into the three pieces xyz, satisfying our three conditions. What if no strings in A are of length at least p? Then our task is even easier because the theorem becomes *vacuously* true: Obviously the three conditions hold for all strings of length at least p if there aren't any such strings.

If s in A has length at least p, consider the sequence of states that M goes through when computing with input s. It starts with q_1 the start state, then goes to, say, q_3, then, say, q_{20}, then q_9, and so on, until it reaches the end of s in state q_{13}. With s in A, we know that M accepts s, so q_{13} is an accept state.

If we let n be the length of s, the sequence of states $q_1, q_3, q_{20}, q_9, \ldots, q_{13}$ has length $n + 1$. Because n is at least p, we know that $n + 1$ is greater than p, the number of states of M. Therefore, the sequence must contain a repeated state. This result is an example of the ***pigeonhole principle***, a fancy name for the rather obvious fact that if p pigeons are placed into fewer than p holes, some hole has to have more than one pigeon in it.

The following figure shows the string s and the sequence of states that M goes through when processing s. State q_9 is the one that repeats.

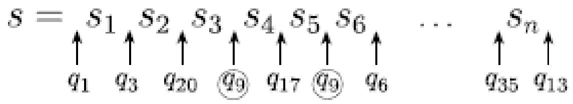


FIGURE 1.71
Example showing state q_9 repeating when M reads s

We now divide s into the three pieces x, y, and z. Piece x is the part of s appearing before q_9, piece y is the part between the two appearances of q_9, and piece z is the remaining part of s, coming after the second occurrence of q_9. So x takes M from the state q_1 to q_9, y takes M from q_9 back to q_9, and z takes M from q_9 to the accept state q_{13}, as shown in the following figure.

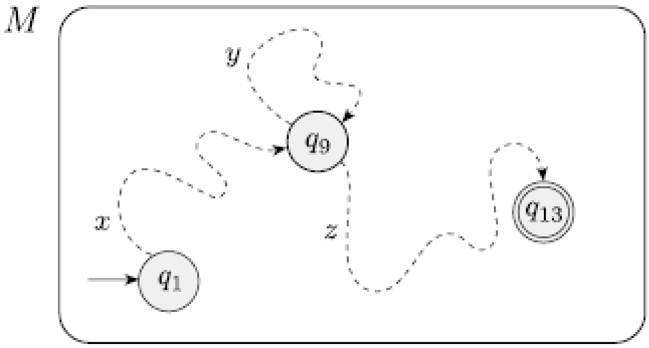


FIGURE 1.72
Example showing how the strings x, y, and z affect M

Let's see why this division of s satisfies the three conditions. Suppose that we run M on input $xyyz$. We know that x takes M from q_1 to q_9, and then the first y takes it from q_9 back to q_9, as does the second y, and then z takes it to q_{13}. With q_{13} being an accept state, M accepts input $xyyz$. Similarly, it will accept xy^iz for any $i > 0$. For the case $i = 0$, $xy^iz = xz$, which is accepted for similar reasons. That establishes condition 1.

Checking condition 2, we see that $|y| > 0$, as it was the part of s that occurred between two different occurrences of state q_9.

In order to get condition 3, we make sure that q_9 is the first repetition in the sequence. By the pigeonhole principle, the first $p+1$ states in the sequence must contain a repetition. Therefore, $|xy| \leq p$.

PROOF Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA recognizing A and p be the number of states of M.

Let $s = s_1s_2 \cdots s_n$ be a string in A of length n, where $n \geq p$. Let $r_1, \ldots, r_{n+1}$ be the sequence of states that M enters while processing s, so $r_{i+1} = \delta(r_i, s_i)$ for $1 \leq i \leq n$. This sequence has length $n + 1$, which is at least $p + 1$. Among the first $p + 1$ elements in the sequence, two must be the same state, by the pigeonhole principle. We call the first of these r_j and the second r_l. Because r_l occurs among the first $p+1$ places in a sequence starting at r_1, we have $l \leq p+1$. Now let $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{l-1}$, and $z = s_l \cdots s_n$.

As x takes M from r_1 to r_j, y takes M from r_j to r_j, and z takes M from r_j to r_{n+1}, which is an accept state, M must accept xy^iz for $i \geq 0$. We know that $j \neq l$, so $|y| > 0$; and $l \leq p+1$, so $|xy| \leq p$. Thus we have satisfied all conditions of the pumping lemma.

To use the pumping lemma to prove that a language B is not regular, first assume that B is regular in order to obtain a contradiction. Then use the pumping lemma to guarantee the existence of a pumping length p such that all strings of length p or greater in B can be pumped. Next, find a string s in B that has length p or greater but that cannot be pumped. Finally, demonstrate that s cannot be pumped by considering all ways of dividing s into x, y, and z (taking condition 3 of the pumping lemma into account if convenient) and, for each such division, finding a value i where $xy^iz \notin B$. This final step often involves grouping the various ways of dividing s into several cases and analyzing them individually. The existence of s contradicts the pumping lemma if B were regular. Hence B cannot be regular.

Finding s sometimes takes a bit of creative thinking. You may need to hunt through several candidates for s before you discover one that works. Try members of B that seem to exhibit the "essence" of B's nonregularity. We further discuss the task of finding s in some of the following examples.

EXAMPLE 1.73

Let B be the language $\{0^n1^n \mid n \geq 0\}$. We use the pumping lemma to prove that B is not regular. The proof is by contradiction.

Assume to the contrary that B is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string 0^p1^p. Because s is a member of B and s has length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in B. We consider three cases to show that this result is impossible.

1. The string y consists only of 0s. In this case, the string $xyyz$ has more 0s than 1s and so is not a member of B, violating condition 1 of the pumping lemma. This case is a contradiction.
2. The string y consists only of 1s. This case also gives a contradiction.
3. The string y consists of both 0s and 1s. In this case, the string $xyyz$ may have the same number of 0s and 1s, but they will be out of order with some 1s before 0s. Hence it is not a member of B, which is a contradiction.

Thus a contradiction is unavoidable if we make the assumption that B is regular, so B is not regular. Note that we can simplify this argument by applying condition 3 of the pumping lemma to eliminate cases 2 and 3.

In this example, finding the string s was easy because any string in B of length p or more would work. In the next two examples, some choices for s do not work so additional care is required. ■

EXAMPLE 1.74

Let $C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$. We use the pumping lemma to prove that C is not regular. The proof is by contradiction.

Assume to the contrary that C is regular. Let p be the pumping length given by the pumping lemma. As in Example 1.73, let s be the string 0^p1^p. With s being a member of C and having length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in C. We would like to show that this outcome is impossible. But wait, it *is* possible! If we let x and z be the empty string and y be the string 0^p1^p, then xy^iz always has an equal number of 0s and 1s and hence is in C. So it *seems* that s can be pumped.

Here condition 3 in the pumping lemma is useful. It stipulates that when pumping s, it must be divided so that $|xy| \leq p$. That restriction on the way that s may be divided makes it easier to show that the string $s = 0^p1^p$ we selected cannot be pumped. If $|xy| \leq p$, then y must consist only of 0s, so $xyyz \notin C$. Therefore, s cannot be pumped. That gives us the desired contradiction.

Selecting the string s in this example required more care than in Example 1.73. If we had chosen $s = (01)^p$ instead, we would have run into trouble because we need a string that *cannot* be pumped and that string *can* be pumped, even taking condition 3 into account. Can you see how to pump it? One way to do so sets $x = \varepsilon$, $y = 01$, and $z = (01)^{p-1}$. Then $xy^iz \in C$ for every value of i. If you fail on your first attempt to find a string that cannot be pumped, don't despair. Try another one!

An alternative method of proving that C is nonregular follows from our knowledge that B is nonregular. If C were regular, $C \cap 0^*1^*$ also would be regular. The reasons are that the language 0^*1^* is regular and that the class of regular languages is closed under intersection, which we proved in footnote 3 (page 46). But $C \cap 0^*1^*$ equals B, and we know that B is nonregular from Example 1.73. ■

EXAMPLE 1.75

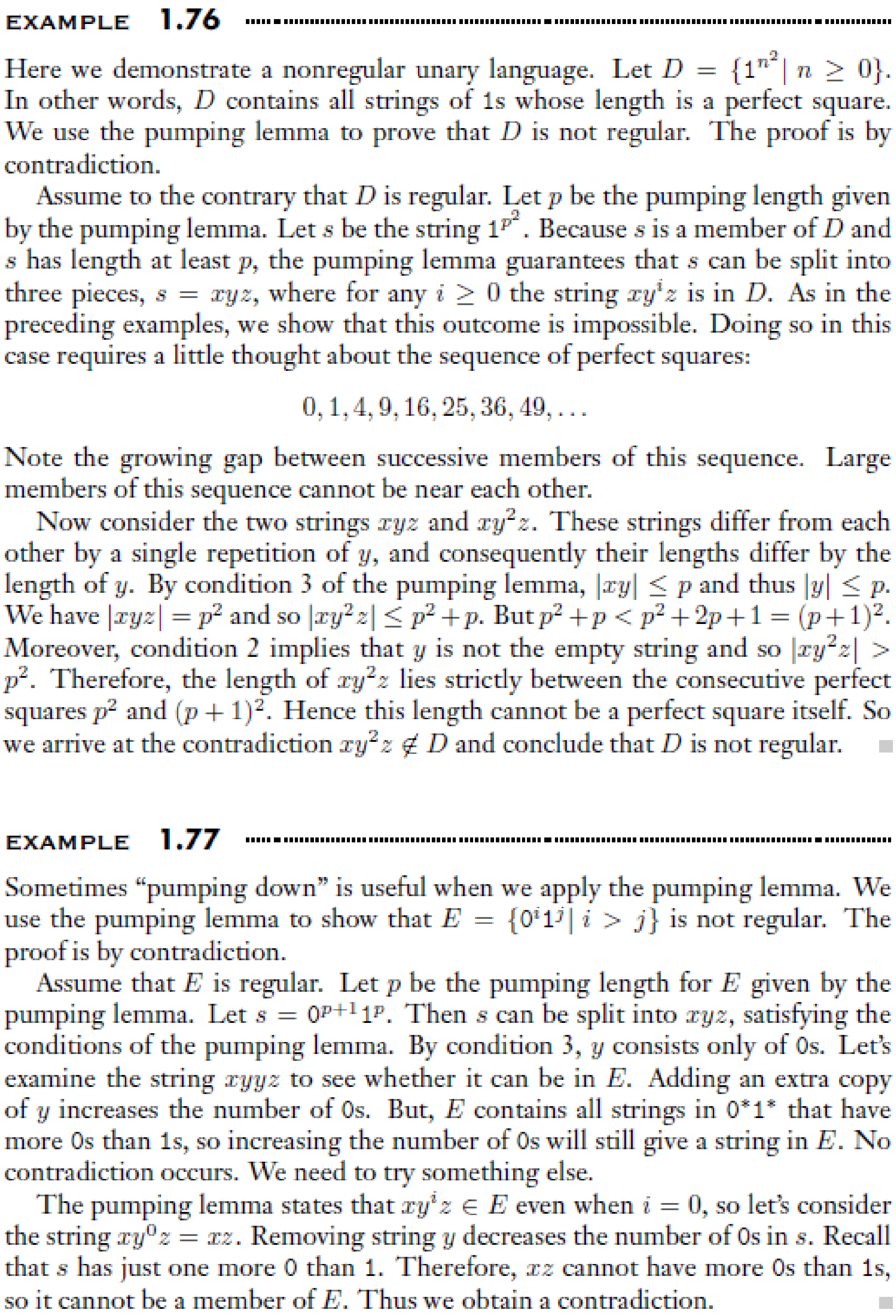
Let $F = \{ww|\ w \in \{0,1\}^*\}$. We show that F is nonregular, using the pumping lemma.

Assume to the contrary that F is regular. Let p be the pumping length given by the pumping lemma. Let s be the string 0^p10^p1. Because s is a member of F and s has length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the three conditions of the lemma. We show that this outcome is impossible.

Condition 3 is once again crucial because without it we could pump s if we let x and z be the empty string. With condition 3 the proof follows because y must consist only of 0s, so $xyyz \notin F$.

Observe that we chose $s = 0^p10^p1$ to be a string that exhibits the "essence" of the nonregularity of F, as opposed to, say, the string 0^p0^p. Even though 0^p0^p is a member of F, it fails to demonstrate a contradiction because it can be pumped.

■



EXAMPLE 1.76

Here we demonstrate a nonregular unary language. Let $D = \{1^{n^2} |\ n \geq 0\}$. In other words, D contains all strings of 1s whose length is a perfect square. We use the pumping lemma to prove that D is not regular. The proof is by contradiction.

Assume to the contrary that D is regular. Let p be the pumping length given by the pumping lemma. Let s be the string 1^{p^2}. Because s is a member of D and s has length at least p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in D. As in the preceding examples, we show that this outcome is impossible. Doing so in this case requires a little thought about the sequence of perfect squares:

$$0, 1, 4, 9, 16, 25, 36, 49, \ldots$$

Note the growing gap between successive members of this sequence. Large members of this sequence cannot be near each other.

Now consider the two strings xyz and xy^2z. These strings differ from each other by a single repetition of y, and consequently their lengths differ by the length of y. By condition 3 of the pumping lemma, $|xy| \leq p$ and thus $|y| \leq p$. We have $|xyz| = p^2$ and so $|xy^2z| \leq p^2 + p$. But $p^2 + p < p^2 + 2p + 1 = (p+1)^2$. Moreover, condition 2 implies that y is not the empty string and so $|xy^2z| > p^2$. Therefore, the length of xy^2z lies strictly between the consecutive perfect squares p^2 and $(p+1)^2$. Hence this length cannot be a perfect square itself. So we arrive at the contradiction $xy^2z \notin D$ and conclude that D is not regular. ■

EXAMPLE 1.77

Sometimes "pumping down" is useful when we apply the pumping lemma. We use the pumping lemma to show that $E = \{0^i1^j |\ i > j\}$ is not regular. The proof is by contradiction.

Assume that E is regular. Let p be the pumping length for E given by the pumping lemma. Let $s = 0^{p+1}1^p$. Then s can be split into xyz, satisfying the conditions of the pumping lemma. By condition 3, y consists only of 0s. Let's examine the string $xyyz$ to see whether it can be in E. Adding an extra copy of y increases the number of 0s. But, E contains all strings in 0^*1^* that have more 0s than 1s, so increasing the number of 0s will still give a string in E. No contradiction occurs. We need to try something else.

The pumping lemma states that $xy^iz \in E$ even when $i = 0$, so let's consider the string $xy^0z = xz$. Removing string y decreases the number of 0s in s. Recall that s has just one more 0 than 1. Therefore, xz cannot have more 0s than 1s, so it cannot be a member of E. Thus we obtain a contradiction. ■

Summary

In this week, we learned about regular languages, expressions and the pumping lemma.