MODULE-2
REGULAR EXPRESSIONS AND LANGUAGES

Regular Expressions:

Regular expressions are another type of language-defining notation. We shall find that regular expressions can define exactly the same languages that the various forms of automata (DFA and NFA) describe: the regular languages. Regular expressions are closely related to NFA and can be thought of as a "user-friendly" alternative to the NFA notation for describing regular languages.

We describe regular language through the notation of R. E. This notation involves a combination of strings or symbols from some alphabet ∑, (,), operators like +, . , *.

In R. E notation we use 3 operations on languages that the operators of R. E represent.

1. **Union**: The union of two regular languages, L_1 and L_2, which are represented using $L_1 \cup L_2$, is also regular and which represents the set of strings that are either in L_1 or L_2 or both.

   Example:

   L_1= {00 , 10, 11, 01} and

   L_2 = {∈ , 100}

   then $L_1 \cup L_2$ = {∈, 00, 10, 11, 01, 100}.

2. Concatenation:

The concatenation of two regular languages, L_1 and L_2, which are represented using L_1. L_2 is also regular and which represents the set of strings that are formed by taking any string in L_1 concatenating it with any string in L_2.

Example:

L_1 = { 0,1 } and L_2 = { 00, 11} then L_1. L_2 = {000, 011, 100, 111}.

3. The closure or star or Kleene closure:

If L is a regular language, then the Kleene closure i.e. L* of L_1 is also regular and represents the set of those strings which are formed by taking a number of strings from L_1 and the same string can be repeated any number of times and concatenating those strings.

Example:

L_1 = { 0,1} then L* is all strings possible with symbols 0 and 1, including a null string.

L*= {∈, 0, 1, 00, 01, 10, 11, 00101, 110011,…….}

Definition of Regular Expressions and the Languages associated with R. E:

BASIS: The basis consists of three parts:

1. The constants ϵ and $\emptyset$ are regular expressions, denoting the languages $\{\epsilon\}$ and $\emptyset$, respectively. That is, $L(\epsilon) = \{\epsilon\}$, and $L(\emptyset) = \emptyset$.
2. If a is any symbol, then **a** is a regular expression. This expression denotes the language $\{a\}$. That is, $L(\mathbf{a}) = \{a\}$. Note that we use boldface font to denote an expression corresponding to a symbol. The correspondence, e.g. that **a** refers to a, should be obvious.
3. A variable, usually capitalized and italic such as L, is a variable, representing any language.

INDUCTION: There are four parts to the inductive step, one for each of the three operators and one for the introduction of parentheses.

1. If E and F are regular expressions, then $E + F$ is a regular expression denoting the union of $L(E)$ and $L(F)$. That is, $L(E+F) = L(E) \cup L(F)$.
2. If E and F are regular expressions, then EF is a regular expression denoting the concatenation of $L(E)$ and $L(F)$. That is, $L(EF) = L(E)L(F)$.
3. If E is a regular expression, then E^* is a regular expression, denoting the closure of $L(E)$. That is, $L(E^*) = \big(L(E)\big)^*$.
4. If E is a regular expression, then (E), a parenthesized E, is also a regular expression, denoting the same language as E. Formally; $L\big((E)\big) = L(E)$.

Precedence of Regular Expression Operators:

Like other algebras, the regular-expression operators have an assumed order of "precedence", which means that operators are associated with their operands in a particular order.

1. The star operator is of highest precedence
2. Next in precedence comes the concatenation or dot operator
3. The union or '+' has least precedency.

Ex: 01* + 1 can be grouped as, (0(1*))+1

Examples:

Finding RE for the given Language:

1.Let L = {w ϵ {a, b }*: |w| is even}.

L = {aa,ab,abba,aabb,ba,baabaa,----------}

Then, RE = ((a + b)(a + b))*

or

RE= (aa+ ab + ba + bb)*

2.Let L = {w ϵ {a, b }*: w starting with string abb}.

L = {abb,abba,abbb,abbab,…….}

Then, RE = abb(a + b)*

3. Let L = {w ϵ {a, b }*: w ending with string abb}.

L = {abb,aabb,babb,ababb---------- }

RE = (a + b)*abb

4. L = {w ϵ {0, 1}* : w have 001 as a substring}.

L = {001,1001,000101,---------- }

RE = (0 + 1)*001(0 + 1)*

5. L = {w ϵ {0, 1}* : w does not have 001 as a substring}.

L = {0,1,010,110,101,-------- }

RE = (1 + 01)*0*

6. L = {w ϵ {a, b}* : w contains an odd number of a's}.

L = {a,aaa,ababa,bbaaaaba---------- }

RE = b*(ab*ab*)* a b* or b*ab*(ab*ab*)*

7. L = {w ϵ {a, b}* :#a(w) mod 3 = 0}.

L = {aaa,abbaba,baaaaaa,…… }

RE = **(b*ab*ab*a)*b***

8. Let L = {w ϵ {a, b }*:#a(w) <= 3}.

L = {a,aa,ba,aaab,bbbabb, ---------- }

RE = b*(a + ε)b*(a + ε)b*(a + ε)b*

9. L = {w ϵ {0, 1}* : w contains no consecutive 0's}

L={0, ε,1,01,10,1010,110,101, ------ }

RE = (0 + ε)(1 + 10)

10. L = {w ϵ {0, 1}* : w contains at least two 0's}

L={00,1010,1100,0001,1010,100,000, ---------}

RE = (0 + 1)*0(0 + 1)*0(0 + 1)*

11. L = { $a^n b^m$ / n>=4 and m<= 3}

RE= (aaaa)a*(ε + b + bb + bbb)

12.L = { $a^n b^m$ / n<=4 and m>= 2}

RE= (ε + a + aa + aaa + aaaa)bb(b)*

13. L = { $a^{2n} b^{2m}$ / n>=0 and m>= 0}

RE= (aa)*(bb)*

14. L = { $a^n b^m$:(m+n) is even}

Since (m+n) is even when both a's and b's are even or both odd.

RE = (aa)*(bb)* + a(aa)*b(bb)*

15. L={w: w ϵ {a, b}* and $n_a(w)$ & $n_b(w)$ is even}

RE=(aa+bb)* + ((ab+ba)(aa+bb)*(ab+ba))*

- For odd number of a's and even number of b's we need one extra a at any position in the above expression.

RE=(aa+bb)* **b** (aa+bb)* +

((ab+ba)(aa+bb)*(ab+ba))* **b** ((ab+ba)(aa+bb)*(ab+ba))*

Finite Automata and Regular Expressions:

We know that DFA, NFA and ε-NFA accepts the same class of languages. In order to show that R. E. defines the same class, we must show that:

1. Every language defined by one of these automata is also defined by a regular expression. For this proof, we can assume the language is accepted by some DFA.
2. Every language defined by a regular expression is defined by one of these automata. For this part of the proof, the easiest is to show that there is an ε-NFA with transitions accepting the same language.

From DFA's to Regular Expressions:

Converting DFA's to Regular Expressions by Eliminating States:

Given a DFA with its set of states and transitions. When we eliminate a state s, all the paths that went through s no longer exist in the automaton. If the language of the automaton is not to change, we must include, on an arc that goes directly from q to p, the labels of paths that went from some state q to state p through s. Since the label of this arc may now involve strings, rather than single symbols and there may even be an infinite number of such strings, we cannot simply list the strings as a label. Fortunately, there is a simple finite way to represent all such strings, use a regular expression.

Method for constructing R. E from FA.

1. For each accepting state $\mathbf{q_f}$ apply the above reduction process to produce an equivalent automaton with RE labels on the arcs. Eliminate all states accept qf and start state q0.
2. If $\mathbf{q_f}$ is not equal to $\mathbf{q_0}$, then we are left with two state automaton.

3. If $\mathbf{q_f} = \mathbf{q_0}$ start state is also final state, then we must perform state elimination so that we are left with one state automata.
4. The desired RE is the sum or union of all the expressions derived from the reduced automaton for each accepting state by rules (2) and (3).

Example:

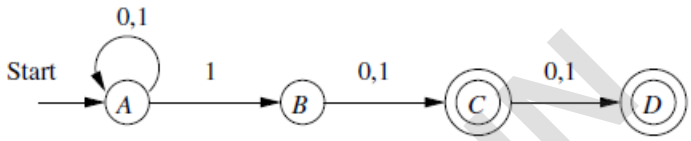


Figure 3.11: An NFA accepting strings that have a 1 either two or three positions from the end

Ans:

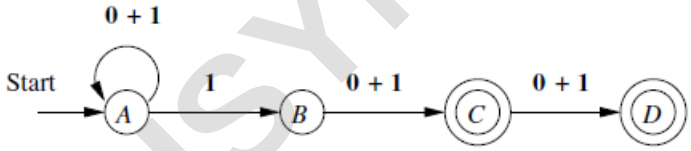


Figure 3.12: The automaton of Fig. 3.11 with regular-expression labels

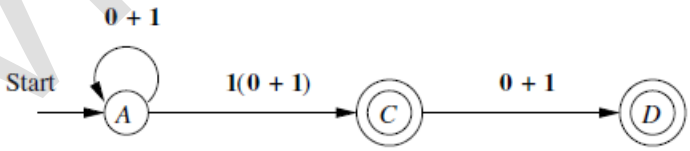


Figure 3.13: Eliminating state B

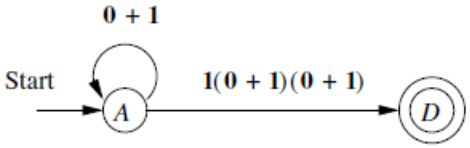


Figure 3.14: A two-state automaton with states A and D

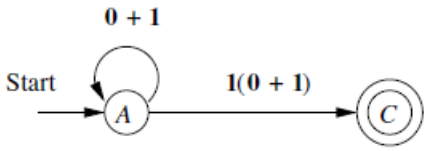


Figure 3.15: Two-state automaton resulting from the elimination of D

Therefore the final RE is: **(0+1)*1(0+1)+(0+1)*1(0+1)(0+1)**

Converting Regular Expressions to Automata:

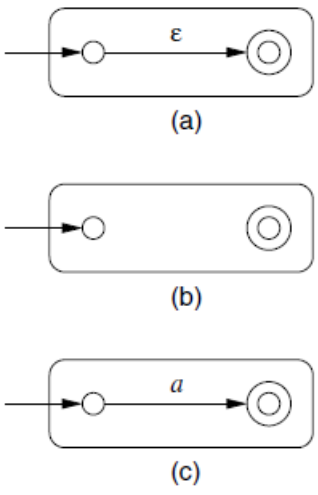
Prove that every language defined by a regular expression is also defined by a finite automaton(6M)

Theorem 3.7: Every language defined by a regular expression is also defined by a finite automaton.

PROOF: Suppose $L = L(R)$ for a regular expression R. We show that $L = L(E)$ for some ϵ-NFA E with:

1. Exactly one accepting state.
2. No arcs into the initial state.
3. No arcs out of the accepting state.

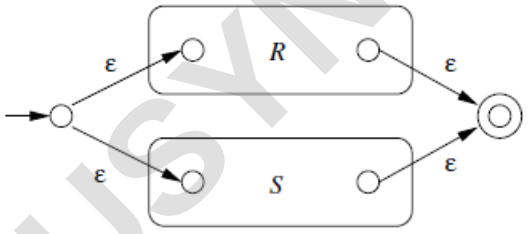
BASIS: There are three parts to the basis, shown in Fig. 3.16. In part (a) we see how to handle the expression ϵ. The language of the automaton is easily seen to be $\{\epsilon\}$, since the only path from the start state to an accepting state is labeled ϵ. Part (b) shows the construction for $\emptyset$. Clearly there are no paths from start state to accepting state, so $\emptyset$ is the language of this automaton. Finally, part (c) gives the automaton for a regular expression **a**. The language of this automaton evidently consists of the one string a, which is also $L(\mathbf{a})$. It



Induction:

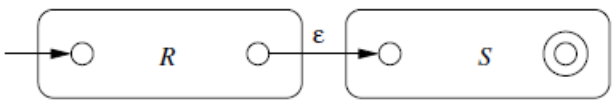
The three parts of the induction are shown in Fig. 3.17. We assume that the statement of the theorem is true for the immediate subexpressions of a given regular expression that is, the languages of these subexpressions are also the languages of ε-NFA's with a single accepting state. The four cases are:

1. The expression is **R+S** for some smaller expressions R and S. Then the automaton of Fig. (a) serves. That is, starting at the new start state, we can go to the start state of either the automaton for R or the automaton for S. We then reach the accepting state of one of these automata, following a path labeled by some string in L(R) or L(S) respectively. Thus, the language of the automaton in **L(R) U L(S)**.



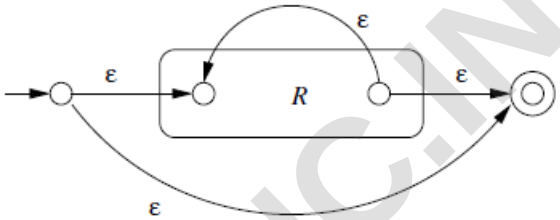
(a)

2. The expression is RS for some smaller expressions R and S. The automaton for the concatenation is shown in Fig. (b). The idea is that the only paths from start to accepting state go first through the automaton for R, where it must follow a path labeled by a string in L(R), and then through the automaton for S, where it follows a path labeled by a string in L(S). Thus, the paths in the automaton of Fig. (b) are all and only those labeled by strings in L(R)L(S).



(b)

3. The expression is R* for some smaller expression R. Then we use the automaton of Fig. (c). That automaton allows us to go either:
   a) Directly from the start state to the accepting state along a path labeled ε. That path lets us accept ε, which is in L(R*) no matter what expression R is.
   b) To the start state of the automaton for R, through that automaton one or more times, and then to the accepting state. This set of paths allows us to accept strings in L(R*).

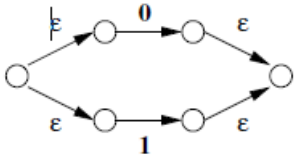


(c)

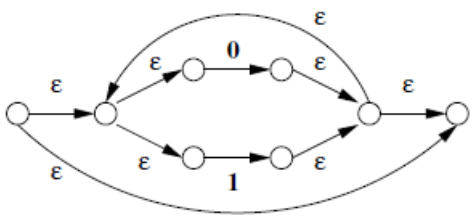
4. The expression is (R) for some smaller expression R. The automaton for R also serves as the automaton for (R), since the parentheses do not change the language defined by the expression.

Example:
Let us convert the regular expression (0+1)* 1(0+1) to an ε-NFA.

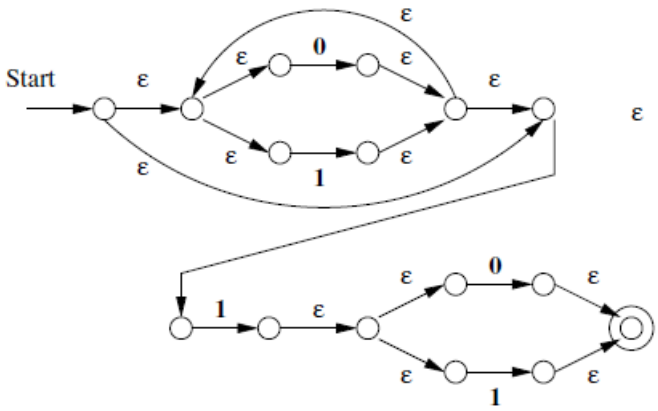
Step 1: Construct an automaton for 0+1.



Step 2: Automaton for (0+1)*



Step 3: Automata for (0+1)* 1(0+1)



Applications of Regular Expressions

A regular expression is the medium of choice for applications that search for patterns in text. The regular expressions are then compiled, behind the scenes, into deterministic or nondeterministic automata, which are then simulated to produce a program that recognizes patterns in text. Two important classes of regular-expression-based applications are: **Lexical analyzers and text search**.

Regular Expressions in UNIX

Before seeing the applications, we shall introduce the UNIX notation for extended regular expressions. This notation gives us a number of additional capabilities.

The UNIX notations for RE involves short-hands that allow complex RE to be written briefly. Most of real applications deal with ASCII character set. The UNIX RE allows us to write character classes to represent large set of characters as briefly as possible.

The rules for character classes are:

1) The symbol . (dot) stands for "any character".
2) $[a_1\ a_2\ a_{3\ldots..}\ a_k] \Rightarrow a_1 + a_2 + a_3 + \ldots.. + a_k$
3) [A-Z]=> all the characters from A to Z.
   [0-9]=>digits ranging from 0 to 9.

- There are special notations for several of the most common classes of characters. For instance:
  a) `[:digit:]` is the set of ten digits, the same as `[0-9]`.[3]
  b) `[:alpha:]` stands for any alphabetic character, as does `[A-Za-z]`.
  c) `[:alnum:]` stands for the digits and letters (alphabetic and numeric characters), as does `[A-Za-z0-9]`.

Operators used in UNIX systems:

1. The operator | is used in place of + to denote union.
2. The operator ? means "zero or one of." Thus, $R?$ in UNIX is the same as $\epsilon + R$ in this book's regular-expression notation.
3. The operator + means "one or more of." Thus, $R+$ in UNIX is shorthand for RR^* in our notation.
4. The operator $\{n\}$ means "n copies of." Thus, $R\{5\}$ in UNIX is shorthand for $RRRRR$.

1) Lexical Analysis

One of the oldest applications of regular expressions was in specifying the component of a compiler called a "lexical analyzer" or "LEX". This component scans the source program and recognizes all tokens, those substrings of consecutive characters that belong together logically. Keywords and identifiers are common examples of tokens, but there are many others.

Example: Lex command to recognize keyword **else, identifier and some relational operators are:**

```
else                    {return(ELSE);}

[A-Za-z][A-Za-z0-9]*    {code to enter the found identifier
                         in the symbol table;
                         return(ID);
                        }

>=                      {return(GE);}

=                       {return(ASGN);}

...
```

Figure 3.19: A sample of `lex` input

2) Finding Patterns in Text

By using regular expression notation, it becomes easy to describe the patterns at a high level, with little effort. Suppose that we want to scan a very large number of Web pages and detect addresses such as street address.

To begin, a street address will probably end in “Street” or its abbreviation “St”. However, some people live on “Avenues” or “Roads” and these might be abbreviated in the address as well. Thus, we might use as the ending for our regular expression something like:

```
Street|St\.|Avenue|Ave\.|Road|Rd\.
```

To include the house number as part of the address, most house numbers are a string of digits. However, some will have a letter following as in "123 Main St,". The regular expression is:

```
'[0-9]+[A-Z]? [A-Z][a-z]*( [A-Z][a-z]*)*
(Street|St\.|Avenue|Ave\.|Road|Rd\.)'
```

Proving Languages Not to Be Regular

Regular languages are the languages accepted by DFA's, by NFA's, by ε-NFA's and also defined by regular expressions.

Not every language is a regular language. Let us consider the language, L={0^n 1^n |n>=1}. This language contains all strings that consist of one or more 0's followed by an equal number of 1's. We claim that L is not a regular language, as it is not possible to track how many 0's we read to match with b's.

The Pumping Lemma for Regular Languages:

(State and Prove Pumping Lemma for regular Languages-6M)

Theorem 4.1: (The *pumping lemma for regular languages*) Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string w in L such that $|w| \geq n$, we can break w into three strings, $w = xyz$, such that:

1. $y \neq \epsilon$.
2. $|xy| \leq n$.
3. For all $k \geq 0$, the string xy^kz is also in L.

That is, we can always find a nonempty string y not too far from the beginning of w that can be "pumped"; that is, repeating y any number of times, or deleting it (the case $k = 0$), keeps the resulting string in the language L.

PROOF: Suppose L is regular. Then $L = L(A)$ for some DFA A. Suppose A has n states. Now, consider any string w of length n or more, say $w = a_1 a_2 \cdots a_m$, where $m \geq n$ and each a_i is an input symbol. For $i = 0, 1, \ldots, n$ define state p_i to be $\hat{\delta}(q_0, a_1 a_2 \cdots a_i)$, where δ is the transition function of A, and q_0 is the start state of A. That is, p_i is the state A is in after reading the first i symbols of w. Note that $p_0 = q_0$.

By the pigeonhole principle, it is not possible for the $n+1$ different p_i's for $i = 0, 1, \ldots, n$ to be distinct, since there are only n different states. Thus, we can find two different integers i and j, with $0 \leq i < j \leq n$, such that $p_i = p_j$. Now, we can break $w = xyz$ as follows:

1. $x = a_1 a_2 \cdots a_i$.
2. $y = a_{i+1} a_{i+2} \cdots a_j$.
3. $z = a_{j+1} a_{j+2} \cdots a_m$.

That is, x takes us to p_i once; y takes us from p_i back to p_i (since p_i is also p_j), and z is the balance of w. The relationships among the strings and states are suggested by Fig. 4.1. Note that x may be empty, in the case that $i = 0$. Also, z may be empty if $j = n = m$. However, y can not be empty, since i is strictly less than j.

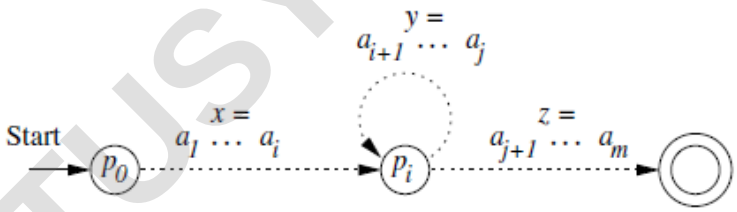


Figure 4.1: Every string longer than the number of states must cause a state to repeat

Now, consider what happens if the automaton A receives the input $xy^k z$ for any $k \geq 0$. If $k = 0$, then the automaton goes from the start state q_0 (which is also p_0) to p_i on input x. Since p_i is also p_j, it must be that A goes from p_i to the accepting state shown in Fig. 4.1 on input z. Thus, A accepts xz.

If $k > 0$, then A goes from q_0 to p_i on input x, circles from p_i to p_i k times on input y^k, and then goes to the accepting state on input z. Thus, for any $k \geq 0$, $xy^k z$ is also accepted by A; that is, $xy^k z$ is in L. □

Applications of the Pumping Lemma:

Basic application of Puming lemma is to prove that language is not regular.

Steps to prove Language is not regular by contradiction method.

1. Assume L is regular and n be the number of states of FA.
2. Apply Pumping Lemma theorem for the given language.
3. Choose a string w, where w€ L and |w| >=n
4. Split w into xyz such that |xy|<=n and y≠ε
5. Choose a value for k such that xy^kz is not in L.
6. Since the result is contradictory to our assumption, the given language is not regular.

(**Refer class work for problems and its solutions**)

Closure Properties of Regular Languages:

In this section, we shall prove several theorems of the form "if certain languages are regular and a language L is formed from them by certain operations, e.g., L is the union of two regular languages, then L is also regular". These theorems are often called **closure properties of the regular languages**.

The principal closure properties for regular languages are:

1. The union of two regular languages is regular.
2. The closure (star) of a regular language is regular.
3. The concatenation of regular languages is regular.
4. The intersection of two regular languages is regular.
5. The complement of a regular language is regular.
6. The difference of two regular languages is regular.
7. The reversal of a regular language is regular.
8. A homomorphism (substitution of strings for symbols) of a regular language is regular.
9. The inverse homomorphism of a regular language is regular.

1. **Regular languages are closed under Union, Concatenation and Star**

   Theorem: If L and M are regular, then LUM, L.M, and L* are also regular.

   Proof: For proof we make use of RE. If L and M are regular languages then there exists regular expressions R1 & R2 such that:

   L=L(R1)

   M=L(R2)

   By the definition of Res, we have

   - R1+R2 is RE denoting the language LUM.
   - R1.R2 is RE denoting the language L.M.
   - R1* is RE denoting the language L*.

Closure Under Complementation

Theorem 4.5: If L is a regular language over alphabet Σ, then $\overline{L} = \Sigma^* - L$ is also a regular language.

PROOF: Let $L = L(A)$ for some DFA $A = (Q, \Sigma, \delta, q_0, F)$. Then $\overline{L} = L(B)$, where B is the DFA $(Q, \Sigma, \delta, q_0, Q - F)$. That is, B is exactly like A, but the accepting states of A have become nonaccepting states of B, and vice versa. Then w is in $L(B)$ if and only if $\hat{\delta}(q_0, w)$ is in $Q - F$, which occurs if and only if w is not in $L(A)$. □

Closure Under Intersection

Theorem: If L and M are regular, then the regular language is closed under intersection.

Proof: Since we already proved the RLs are closed under union and Complementation, we can obtain the intersection of languages L and M by the identity (as per one of DeMorgan's law, we can obtain intersection of two sets in terms of complement of union of complement of languages):

$$L \cap M = \overline{\overline{L} \cup \overline{M}}$$

.Thus Regular languages are closed under intersection.

Closure Under Difference

In terms of languages, L-M, the difference of L and M, is the set of strings that are in language L but not in language M. The regular languages are also closed under this operation, and the proof follows easily from the theorems just proven.

Theorem 4.10: If L and M are regular languages, then so is $L - M$.

PROOF: Observe that $L - M = L \cap \overline{M}$. By Theorem 4.5, $\overline{M}$ is regular, and by Theorem 4.8 $L \cap \overline{M}$ is regular. Therefore $L - M$ is regular. □

Reversal:

The *reversal* of a string $a_1 a_2 \cdots a_n$ is the string written backwards, that is, $a_n a_{n-1} \cdots a_1$. We use w^R for the reversal of string w. Thus, 0010^R is 0100, and $\epsilon^R = \epsilon$.

The reversal of a language L, written L^R, is the language consisting of the reversals of all its strings. For instance, if $L = \{001, 10, 111\}$, then $L^R = \{100, 01, 111\}$.

Given a language L that is $L(A)$ for some finite automaton, perhaps with nondeterminism and ϵ-transitions, we may construct an automaton for L^R by:

1. Reverse all the arcs in the transition diagram for A.
2. Make the start state of A be the only accepting state for the new automaton.
3. Create a new start state p_0 with transitions on ϵ to all the accepting states of A.

The result is an automaton that simulates A "in reverse," and therefore accepts a string w if and only if A accepts w^R. Now, we prove the reversal theorem formally.

Theorem 4.11 : If L is a regular language, so is L^R.

PROOF: Assume L is defined by regular expression E. The proof is a structural induction on the size of E. We show that there is another regular expression E^R such that $L(E^R) = \big(L(E)\big)^R$; that is, the language of E^R is the reversal of the language of E.

BASIS: If E is ϵ, $\emptyset$, or $\mathbf{a}$, for some symbol a, then E^R is the same as E. That is, we know $\{\epsilon\}^R = \{\epsilon\}$, $\emptyset^R = \emptyset$, and $\{a\}^R = \{a\}$.

INDUCTION: There are three cases, depending on the form of E.

1. $E = E_1 + E_2$. Then $E^R = E_1^R + E_2^R$. The justification is that the reversal of the union of two languages is obtained by computing the reversals of the two languages and taking the union of those languages.

2. $E = E_1E_2$. Then $E^R = E_2^R E_1^R$. Note that we reverse the order of the two languages, as well as reversing the languages themselves. For instance, if $L(E_1) = \{01, 111\}$ and $L(E_2) = \{00, 10\}$, then $L(E_1E_2) = \{0100, 0110, 11100, 11110\}$. The reversal of the latter language is

$$\{0010, 0110, 00111, 01111\}$$

   If we concatenate the reversals of $L(E_2)$ and $L(E_1)$ in that order, we get

$$\{00, 01\}\{10, 111\} = \{0010, 00111, 0110, 01111\}$$

   which is the same language as $\big(L(E_1E_2)\big)^R$. In general, if a word w in $L(E)$ is the concatenation of w_1 from $L(E_1)$ and w_2 from $L(E_2)$, then $w^R = w_2^R w_1^R$.

3. $E = E_1^*$. Then $E^R = (E_1^R)^*$. The justification is that any string w in $L(E)$ can be written as $w_1w_2 \cdots w_n$, where each w_i is in $L(E)$. But

$$w^R = w_n^R w_{n-1}^R \cdots w_1^R$$

   Each w_i^R is in $L(E^R)$, so w^R is in $L\big((E_1^R)^*\big)$. Conversely, any string in $L\big((E_1^R)^*\big)$ is of the form $w_1w_2 \cdots w_n$, where each w_i is the reversal of a string in $L(E_1)$. The reversal of this string, $w_n^R w_{n-1}^R \cdots w_1^R$, is therefore a string in $L(E_1^*)$, which is $L(E)$. We have thus shown that a string is in $L(E)$ if and only if its reversal is in $L\big((E_1^R)^*\big)$.

Homomorphisms

A string homomorphism is a function on strings that works by substituting a particular string for each symbol.

Formally, if h is a homomorphism on alphabet Σ, and $w = a_1a_2\cdots a_n$ is a string of symbols in Σ, then $h(w) = h(a_1)h(a_2)\cdots h(a_n)$. That is, we apply h to each symbol of w and concatenate the results, in order. For instance, if h is the homomorphism in Example 4.13, and $w = 0011$, then $h(w) = h(0)h(0)h(1)h(1) = (ab)(ab)(\epsilon)(\epsilon) = abab$, as we claimed in that example.

Theorem 4.14: If L is a regular language over alphabet Σ, and h is a homomorphism on Σ, then $h(L)$ is also regular.

PROOF: Let $L = L(R)$ for some regular expression R. In general, if E is a regular expression with symbols in Σ, let $h(E)$ be the expression we obtain by replacing each symbol a of Σ in E by $h(a)$. We claim that $h(R)$ defines the language $h(L)$.

The proof is an easy structural induction that says whenever we take a subexpression E of R and apply h to it to get $h(E)$, the language of $h(E)$ is the same language we get if we apply h to the language $L(E)$. Formally, $L\big(h(E)\big) = h\big(L(E)\big)$.

BASIS: If E is ϵ or $\emptyset$, then $h(E)$ is the same as E, since h does not affect the string ϵ or the language $\emptyset$. Thus, $L\big(h(E)\big) = L(E)$. However, if E is $\emptyset$ or ϵ, then $L(E)$ contains either no strings or a string with no symbols, respectively. Thus $h\big(L(E)\big) = L(E)$ in either case. We conclude $L\big(h(E)\big) = L(E) = h\big(L(E)\big)$.

The only other basis case is if $E = \mathbf{a}$ for some symbol a in Σ. In this case, $L(E) = \{a\}$, so $h\big(L(E)\big) = \{h(a)\}$. Also, $h(E)$ is the regular expression that is the string of symbols $h(a)$. Thus, $L\big(h(E)\big)$ is also $\{h(a)\}$, and we conclude $L\big(h(E)\big) = h\big(L(E)\big)$.

INDUCTION: There are three cases, each of them simple. We shall prove only the union case, where $E = F+G$. The way we apply homomorphisms to regular expressions assures us that $h(E) = h(F + G) = h(F) + h(G)$. We also know that $L(E) = L(F) \cup L(G)$ and

$$L\big(h(E)\big) = L\big(h(F) + h(G)\big) = L\big(h(F)\big) \cup L\big(h(G)\big) \tag{4.2}$$

by the definition of what "+" means in regular expressions. Finally,

$$h\big(L(E)\big) = h\big(L(F) \cup L(G)\big) = h\big(L(F)\big) \cup h\big(L(G)\big) \tag{4.3}$$

because h is applied to a language by application to each of its strings individually. Now we may invoke the inductive hypothesis to assert that $L\big(h(F)\big) = h\big(L(F)\big)$ and $L\big(h(G)\big) = h\big(L(G)\big)$. Thus, the final expressions in (4.2) and

(4.3) are equivalent, and therefore so are their respective first terms; that is, $L(h(E)) = h(L(E))$.

Inverse Homomorphisms:

Homomorphisms may also be applied "backwards," and in this mode they also preserve regular languages. That is, suppose h is a homomorphism from some alphabet Σ to strings in another (possibly the same) alphabet T.[2] Let L be a language over alphabet T. Then $h^{-1}(L)$, read "h inverse of L," is the set of strings w in Σ^* such that $h(w)$ is in L. Figure 4.5 suggests the effect of a homomorphism on a language L in part (a), and the effect of an inverse homomorphism in part (b).

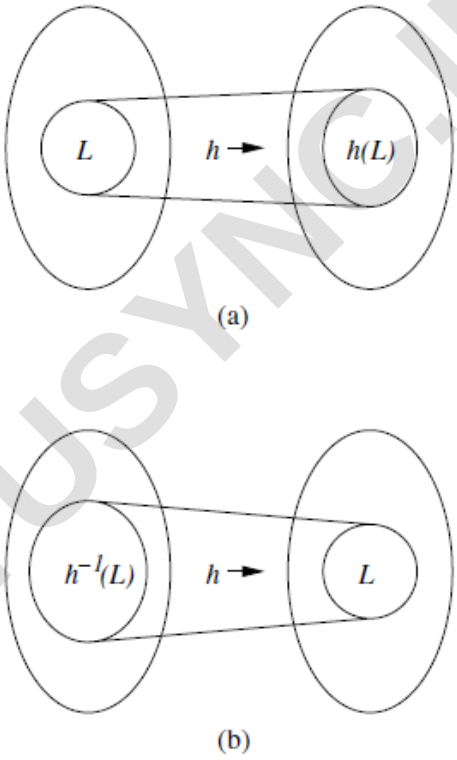


Figure 4.5: A homomorphism applied in the forward and inverse direction

Theorem 4.16: If h is a homomorphism from alphabet Σ to alphabet T, and L is a regular language over T, then $h^{-1}(L)$ is also a regular language.

PROOF: The proof starts with a DFA A for L. We construct from A and h a DFA for $h^{-1}(L)$ using the plan suggested by Fig. 4.6. This DFA uses the states of A but translates the input symbol according to h before deciding on the next state.

Formally, let L be $L(A)$, where DFA $A = (Q, T, \delta, q_0, F)$. Define a DFA

$$B = (Q, \Sigma, \gamma, q_0, F)$$

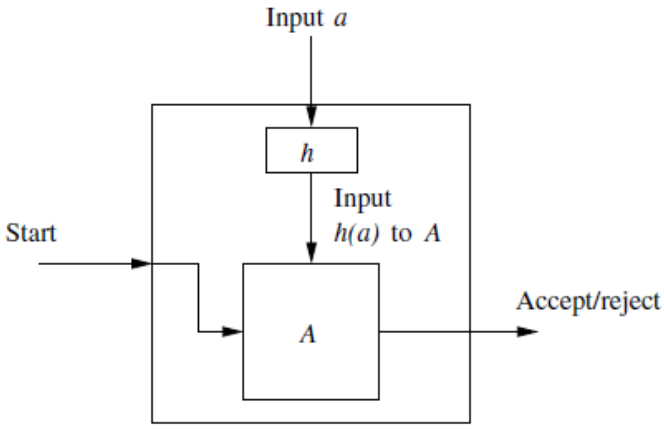


Figure 4.6: The DFA for $h^{-1}(L)$ applies h to its input, and then simulates the DFA for L

where transition function γ is constructed by the rule $\gamma(q,a) = \hat{\delta}\big(q, h(a)\big)$. That is, the transition B makes on input a is the result of the sequence of transitions that A makes on the string of symbols $h(a)$. Remember that $h(a)$ could be ϵ, it could be one symbol, or it could be many symbols, but $\hat{\delta}$ is properly defined to take care of all these cases.

It is an easy induction on $|w|$ to show that $\hat{\gamma}(q_0, w) = \hat{\delta}\big(q_0, h(w)\big)$. Since the accepting states of A and B are the same, B accepts w if and only if A accepts $h(w)$. Put another way, B accepts exactly those strings w that are in $h^{-1}(L)$.

Equivalence and Minimization of Automata:

Testing Equivalence of States:

We say that states p and q are **equivalent** if:

> For all input strings w, $\hat{\delta}(p, w)$ is an accepting state if and only if $\hat{\delta}(q, w)$ is an accepting state.

If two states are not equivalent, then we say they are distinguishable. That

is, state p is distinguishable from state q if there is at least one string w such that one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting, and the other is not accepting.

To find states that are equivalent, we make our best efforts to find pairs of states that are distinguishable. The algorithm, which we refer to as the **table-filling**

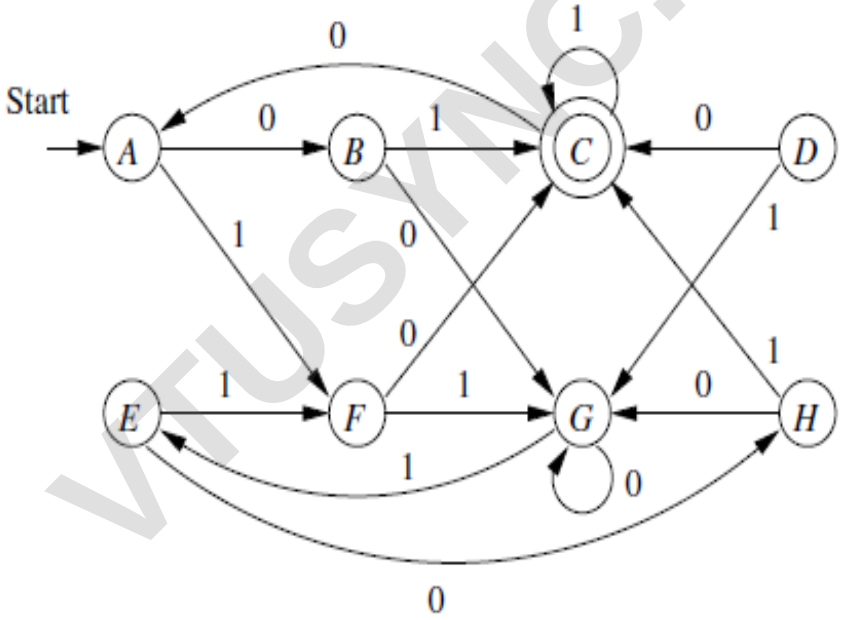
algorithm, is a recursive discovery of distinguishable pairs in a DFA A=(Q, ∑, δ, q_0, F).

Table-filling algorithm:

BASIS: If p is an accepting state and q is nonaccepting, then the pair $\{p, q\}$ is distinguishable.

INDUCTION: Let p and q be states such that for some input symbol a, $r = \delta(p, a)$ and $s = \delta(q, a)$ are a pair of states known to be distinguishable. Then $\{p, q\}$ is a pair of distinguishable states. The reason this rule makes sense is that there must be some string w that distinguishes r from s; that is, exactly one of $\hat{\delta}(r, w)$ and $\hat{\delta}(s, w)$ is accepting. Then string aw must distinguish p from q, since $\hat{\delta}(p, aw)$ and $\hat{\delta}(q, aw)$ is the same pair of states as $\hat{\delta}(r, w)$ and $\hat{\delta}(s, w)$.

Example: Consider the following DFA,



Let us execute the table-filling algorithm on the DFA,

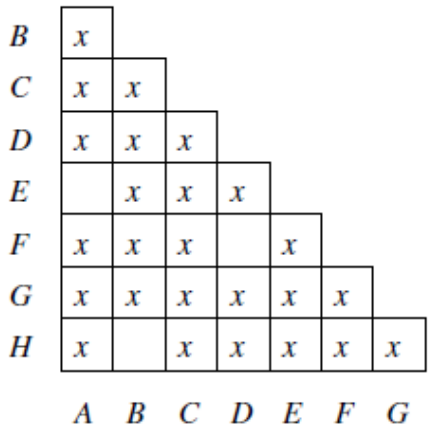


Figure 4.9: Table of state inequivalences

Minimization of DFA's:

An important consequence of the test for equivalence of states is that we can "minimize" DFA's. That is, for each DFA we can find an equivalent DFA that has as few states as any DFA accepting the same language. Moreover, except for our ability to call the states by whatever names we choose, this minimum-state DFA is unique for the language. The algorithm is as follows:

1.First, eliminate any state that cannot be reached from the start state.

2. Then, partition the remaining states into blocks, so that all states in the same block are equivalent, and no pair of states from different blocks are equivalent.

Consider the table of Fig. 4.9, where we determined the state equivalences and distinguishabilities for the states of Fig. 4.8. The partition of the states into equivalent blocks is {A, E}, {B, H}, {C}, {D, F}, {G}.

Note: **The equivalence of states is transitive**. That is, if in some DFA A=(Q, ∑, δ, qo, F) we find that states p and q are equivalent, and we also find that q and r are equivalent, then it must be that p and r are equivalent.

We are now able to state succinctly the algorithm for minimizing a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

1. Use the table-filling algorithm to find all the pairs of equivalent states.

2. Partition the set of states Q into blocks of mutually equivalent states by the method described above.

3. Construct the minimum-state equivalent DFA B by using the blocks as its states. Let γ be the transition function of B. Suppose S is a set of equivalent states of A, and a is an input symbol. Then there must exist one block T of states such that for all states q in S, $\delta(q, a)$ is a member of block T. For if not, then input symbol a takes two states p and q of S to states in different blocks, and those states are distinguishable by Theorem 4.24. That fact lets us conclude that p and q are not equivalent, and they did not both belong in S. As a consequence, we can let $\gamma(S, a) = T$. In addition:

   (a) The start state of B is the block containing the start state of A.

   (b) The set of accepting states of B is the set of blocks containing accepting states of A. Note that if one state of a block is accepting, then all the states of that block must be accepting. The reason is that any accepting state is distinguishable from any nonaccepting state, so you can't have both accepting and nonaccepting states in one block of equivalent states.

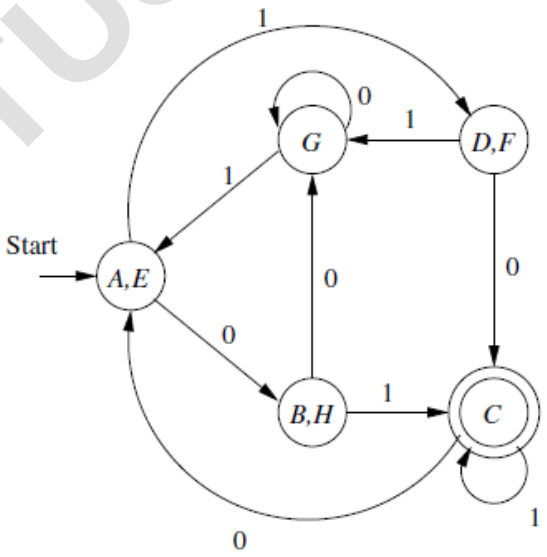


Figure 4.12: Minimum-state DFA equivalent to Fig. 4.8

****