Unit-2

FINITE AUTOMATA, REGULAR EXPRESSIONS: An application of finite automata; Finite automata with Epsilon-transitions; Regular expressions; Finite Automata and Regular Expressions; Applications of Regular Expressions.

Application of finite automata

Finding Strings in Text

A common problem in the age of the Web and other on-line text repositories is the following, given a set of words, find all documents that contain one (or all) of those words. A search engine is a popular example of this process. The search engine uses a particular technology, called *inverted indexes,* where for each word appearing on the Web (there are 100,000,000 different words), a list of all the places where that word occurs is stored.

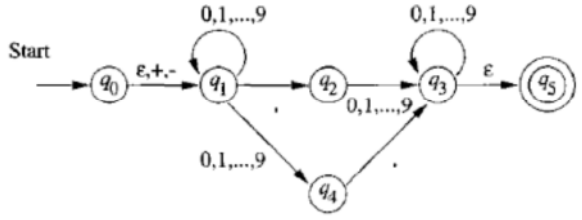
The documents to be searched cannot be cataloged. For example: Amazon.com does not make it easy for crawlers to find all the pages for all the books that the company sells. Rather, these pages are generated "'on the fly" in response to queries.

Finite Automata With Epsilon-Transitions

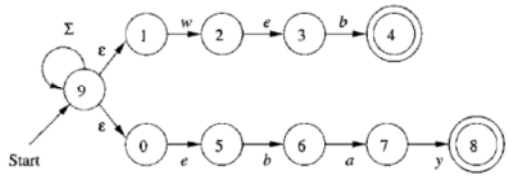
This is another extension of the finite automaton. The new "feature" is that we allow a transition on e, the empty string. In effect, an NFA is allowed to make a transition spontaneously, without receiving an input symbol. Like the non determinism, this new capability does not expand the class of languages that can be accepted by finite automata, but it does give us some added "programming convenience."

Uses of e-Transitions

Example Given an e-NFA that accepts decimal numbers consisting of:
1. An optional + or — sign,
2. A string of digits,
3. A decimal point, and
4. Another string of digits. Either this string of digits or the string (2) can be empty, but at least one of the two strings of digits must be nonempty



Example: Using e transitions to help recognize keywords



The Formal Notation for an e-NFA

We may represent an e-NFA exactly as we do an NFA, with one exception: the transition function must include information about transitions on t. Formally, we represent an e-NFA *A* by *A=(Q, ∑ ,q0,F,δ),* where all components have their same interpretation as for an NFA, except thatδ is now a function that takes as arguments:

1. A state in Q, and
2. A member of ∑ U {e}, that is, either an input symbol, or the symbol e.

We require that e. the symbol for the empty string, cannot be a member of the alphabet so no confusion results.

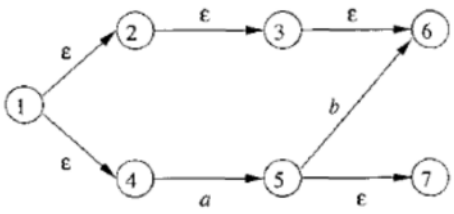
Epsilon-Closures

We shall proceed to give formal definitions of an extended transition function for e-NFA's, which leads to the definition of acceptance of st rings and languages by these automata, and eventually lets us explain why e-NFA:s can be simulated by DFA's. However, we first need to learn a central definition, called the *e-closure* of a state. Informally, wo f close a state *q* by following all transitions out of *q* that are labeled e. However, when we get to other states by following c, we follow the e-transitions out of those states, and so on, eventually finding every state that can be reached from *q* along any path whose arcs are all labeled e. Formally, we define the e-closure ECLOSE(Q) recursively, as follows:

BASIS: State q is in ECLOSE(q).

INDUCTION: If state *p* is in ECLOSE(q), and there is a transition from state *p* to state *r* labeled e, then *r* is in ECLOSE(q). More precisely, if δ is the transition function of the e-NFA involved, and *p* is in ECLOSE(q), then ECLOSE(q) also contains all the states δ(p, e).

Example



For the above given automaton , each state is its own e-closure, with two exceptions: ECLOSE(q0) = {q0,q1) and ECLOSE(q3) = {q3,q5}- The reason is that there are only two e-transitions, one that adds qi to ECLOSE(q0) and the other that adds q5 to ECLOSE(q3) ECLOSE(1) = {1,2,3,4,6}

Extended Transitions and Languages for e-NFA's

The e-closure allows us to explain easily what the transitions of an e-NFA look like when given a sequence of (non-e) inputs. Suppose that $E = (Q, \sum, \delta. qo, F)$ is an e-NFA. We first define δ', the extended transition function, to reflect what happens on a sequence of inputs. The intent is that δ' (q. w) is the set of states that can be reached along a path whose labels, when concatenated, form the string w. As always, e's along this path do not contribute to w. The appropriate recursive definition of δ' is:

B A S I S : δ' *(q, e)* = ECLOSE(q), That is, if the label of the path is e, then we can follow only e-labeled arcs extending from state *q:* that is exactly what ECLOSE does.

INDUCTION: Suppose w is of the form xa, where a is the last symbol of w. Note that a is a member of ∑; it cannot be e, which is not in ∑. We compute δ'*(q, w)* as follows:

1. Let {p1 ,p2….pk} be δ'*(q,* a;). That is, the pi's are all and only the states that we can reach from *q* following a path labeled *x.* This path may end with one or more transitions labeled e. and may have other *e*- transit ions, as well.
2. Let $U_{i=1}$ δ(pi, a) be the set {r1, r2…..rm }. That is, follow all transitions labeled *a* from states we can reach from *q* along paths labeled *x.* The rj's are *some* of the states we can reach from *q* along paths labeled *w.* The additional states we can reach are found from the rj's by following e-labeled arcs in step (3), below.
3. Then δ'*(q,w)=*$U_{j=1}$ ECI.OSE (rj). This additional closure step includes all the paths from *q* labeled *w,* by considering the possibility that there arc additional e-labeled arcs that we can follow after making a transition on the final "real" symbol, a.
   Regular Expressions and Languages

Regular Expressions

Definition: A regular expression is recursively defined as follows.

1. ϕ is a regular expression denoting an empty language.
2. ε-(epsilon) is a regular expression indicates the language containing an empty string.
3. *a* is a regular expression which indicates the language containing only {a}
4. If R is a regular expression denoting the language L_R and S is a regular expression denoting the language L_S, then
   a. R+S is a regular expression corresponding to the language $L_R U L_S$.
   b. R.S is a regular expression corresponding to the language $L_R.L_S$..
   c. R* is a regular expression corresponding to the language L_R^*.
5. The expressions obtained by applying any of the rules from 1-4 are regular expressions.

Regular expressions	Meaning
(a+b)*	Set of strings of a's and b's of any length including the NULL string.
(a+b)*abb	Set of strings of a's and b's ending with the string abb
ab(a+b)*	Set of strings of a's and b's starting with the string ab.
(a+b)*aa(a+b)	Set of strings of a's and b's having a sub string

*	aa.
a*b*c*	Set of string consisting of any number of a's(may be empty string also) followed by any number of b's(may include empty string) followed by any number of c's(may include empty string).
$a^+b^+c^+$	Set of string consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'.
aa*bb*cc*	Set of string consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'.
(a+b)* (a + bb)	Set of strings of a's and b's ending with either *a* or *bb*
(aa)*(bb)*b	Set of strings consisting of even number of a's followed by odd number of b's
(0+1)*000	Set of strings of 0's and 1's ending with three consecutive zeros(or ending with 000)
(11)*	Set consisting of even number of 1's

The Operators of Regular Expressions

1. The *union* of two languages *L* and M, denoted *L* U *M,* is the set of strings that are in either *L or M,* or both. For example, if *L*— {001,10,111} and *M* = {e, 001}, then *LUM* = {e, 10,001, 111}.
2. The *concatenation* of languages *L* and *M* is the set of strings that can be formed by taking any string in *L* and concatenating it with any string in M. For example, if *L* = {001,10,111} and *M* = {e,001}, then L.M, or just LM, is {001,10, 111, 001001,10001,111001}. The first three strings in *LM* are the strings in *L* concatenated with e. Since e is the identity for concatenation, the resulting strings are the same as the strings of *L.* However, the last three strings in *LM* are formed by taking each string in *L* and concatenating it with the second string in M, which is 001. For instance, 10 from *L* concatenated with 001 from *M* gives us 10001 for *LM.*
3. The *closure* (or *star,* or *Kleene closure)1* of a language *L* is denoted *L** and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (i.e., the same string may be selected more than once) and concatenating all of them. For instance, if *L*= {0,1}, then L* is all strings of O's and l's. If *L* = {0,11}, then L* consists of those strings of 0's and l's such that the l's come in pairs, e.g., 011, 11110, and e, but not 01011 or 101.

Building Regular Expressions

1. If *E* and *F* are regular expressions, then *E + F i*Is a regular expression denoting the union of *L(E)* and *L(F).* That is, *L(E + F) = L{E)* U *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).*
4. If *E* is a regular expression, then (E), a parenthesized *E,* is also a regular expression, denoting the same language as *E*.

Precedence of Regular-Expression Operators

1. The star operator is of highest precedence.
2. Next in precedence comes the concatenation or "dot" operator.
3. Finally, all unions (+ operators) are grouped with their operands.

Finite Automata and Regular Expressions

Every language defined by one of these automata is also defined by a regular expression.
Every language defined by a regular expression is defined by one of these automata.

From DFA's to Regular Expressions

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}\left(R_{kk}^{(k-1)}\right)^{*}R_{kj}^{(k-1)}$$

Applications of Regular Expressions

1. Regular Expressions in UNIX
2. Lexical Analysis
3. Finding Patterns in Text