Course Outcomes

CO 1: Acquire fundamental understanding of the core concepts in automata theory and Theory of Computation

CO 2: Design and develop lexical analyzers, parsers and code generators

CO 3: Design Grammars and Automata (recognizers) for different language classes and become knowledgeable about restricted models of Computation (Regular, Context Free) and their relative powers.

CO 4: Acquire fundamental understanding of the structure of a Compiler and Apply concepts automata theory and Theory of Computation to design Compilers

CO 5: Design computations models for problems in Automata theory and adaptation of such model in the field of compilers

Institution Vision

To produce top-quality engineers who are groomed for attaining excellence in their profession and competitive enough to help in the growth of nation and global society.

Institution Mission

M1: To offer affordable high-quality graduate program in engineering with value education and make the students socially responsible.

M2: To support and enhance the institutional environment to attain research excellence in both faculty and students and to inspire them to push the boundaries of knowledge base.

M3: To identify the common areas of interest amongst the individuals for the effective industry- institute partnership in a sustainable way by systematically working together.

M4: To promote the entrepreneurial attitude and inculcate innovative ideas among the engineering professionals.

Department Vision

To be a center of excellence in Information Science & Engineering education, research and training to meet the growing needs of the industry and society.

Department Mission

M1: To impart theoretical and practical knowledge through the concepts and technologies in Information Science and Engineering

M2: To foster research, collaboration and higher education with premier institutions and industries.

M3: Promote innovation and entrepreneurship to fulfill the needs of the society and industry

Program Educational Objectives

PEO1: Analyse, design and implement solutions to the real-world problems in the field of Information Science and Engineering with multidisciplinary setup

PEO2: Keep abreast with the technology, innovation and pursue higher education with high standards of social and professional ethics

PEO3: Develop professional and entrepreneurship skills to work effectively as an individual and in a team to meet the ever-changing goals of the organization

Program Outcomes

PO1: **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

PO2: **Problem Analysis:** Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural science and engineering sciences.

PO3: **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal and environmental considerations.

PO4: **Conduct investigations of complex problems:** Use research based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: **Modern tool usage:** Create, select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations

PO6: **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice

PO7: **Environment sustainability:** Understand the impact of the professional engineering solutions in the societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions

PO11: **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to ones own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: **Lifelong learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broader context of technological change.

Program Specific Outcomes

PSO1: Design, implement and maintain the information systems that fulfill the current needs of the industry and society

PSO2: Apply computational theory, storage and networking concepts to solve the day to day problems of the world

Regular expression

Regular expressions also may be thought of as a "programming language", in which we express some important applications such as text search applications or compiler components. Regular expressions are closely related to nondeterministic finite automata and can be thought of as a "user friendly" alternative to the NFA notation for describing software components.

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 $L_R \cup L_S$.
   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.

The table shows some examples of regular expressions and the language corresponding to theseregular 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'sfollowed 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

Example: Obtain a regular expression to accept a language consisting of strings of a's and b's of even length.

String of a's and b's of even length can be obtained by the combination of the strings aa, ab, ba and bb. The language may even consist of an empty string denoted expression can be of the form by ε. So, the regular (aa + ab + ba + bb)* The * closure includes the empty string.

Note: This regular expression can also be represented using set notation as

L(R) = {(aa + ab + ba + bb)n | n ≥ 0}

Example: Obtain a regular expression to accept a language consisting of strings of a's and b's of odd length.

String of a's and b's of odd length can be obtained by the combination of the strings aa, ab, ba and bb followed by either a or b. So, the regular expression can be of the form

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

String of a's and b's of odd length can also be obtained by the combination of the strings aa, ab, ba and bb preceded by either a or b. So, the regular expression can also be represented as

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

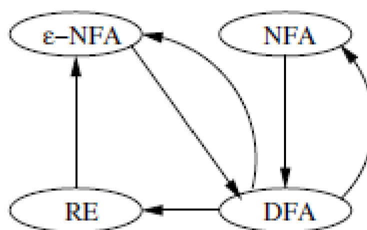
Note: Even though these two expression are seems to be different, the language corresponding to those two expression is same. So, a variety of regular expressions can be obtained for a language and all are equivalent.

Precedence of Regular Expression Operators

- The star operator is of highest precedence. That is, it applies only to the smallest sequence of symbols to its left that is a well formed regular expression.
- Next in precedence comes the concatenation or "dot" operator. After grouping all-stars to their operands, we group concatenation operators to their operands. That is, all expressions that are juxtaposed (adjacent, with no intervening operator) are grouped together. Since concatenation is an associative operator it does not matter in what order we group consecutive concatenations, although if there is a choice to be made, you should group them from the left. For instance, 012 is grouped (01)2.
- Finally, all unions (+ operators) are grouped with their operands. Since union is also associative, it again matters little in which order consecutive unions are grouped, but we shall assume grouping from the left.

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.



Obtain Finite Automata from the regular expression

Theorem: Let R be a regular expression. Then there exists a finite automaton M = (Q, ∑, δ,q_0, A) which accepts L(R).

Proof: By definition, ε, φ and *a* are regular expressions. So, the corresponding machines to recognize these expressions are shown in figure a, b and c respectively.

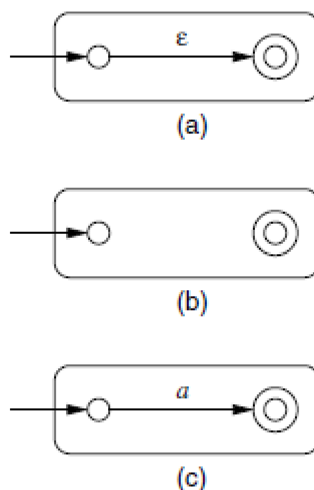


Fig NFAs to accept ε , φ and a

The schematic representation of a regular expression R to accept the language L(R) is shown in figure. where q is the start state and f is the final state of machine M.

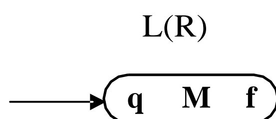


Fig: Schematic representation of FA accepting L(R)

In the definition of a regular expression it is clear that if R and S are regular expression, then R+S and R.S and R* are regular expressions which clearly uses three operators '+', '.' and '*'. Let us take each case separately and construct equivalent machine. Let $M_1 = (Q_1, \sum_1, \delta_1, q_1, f_1)$ be a machine which accepts the language $L(R_1)$ corresponding to the regular expression R_1. Let $M_2 = (Q_2, \sum_2, \delta_2, q_2, f_2)$ be a machine which accepts the language $L(R_2)$ corresponding to the regular expression R_2.

Case 1: R = R + S. We can construct an NFA which accepts either L(R) or L(S) which can be represented as L(R + S) as shown in figure.

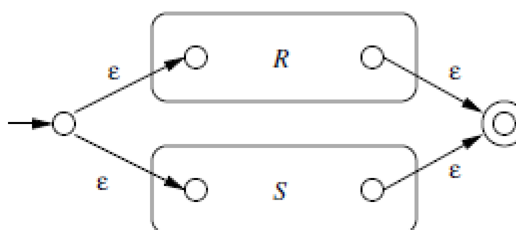


Fig. To accept the language L(R + S)

It is clear from figure that the machine can either accept L(R) or L(S). Here, q_0 is the start state of the combined machine and q_f is the final state of combined machine M.

Case 2: R = R **.** S. We can construct an NFA which accepts L(R) followed by L(S) which can be represented as L(R **.** S) as shown in figure.

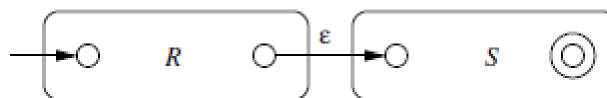


Fig. To accept the language L(R . S)

It is clear from figure that the machine after accepting L(R) moves from state q1 to f1.

Since there is a ε-transition, without any input there will be a transition from state f1 to state q2. In state q2, upon accepting L(S), the machine moves to f2 which is the final state. Thus, q1 which is the start state of machine M1 becomes the start state of the combined machine M and f2 which is the final state of machine M2, becomes the final state of machine M and accepts the language L(R.S).

Case 3: R = $(R)^*$. We can construct an NFA which accepts either $L(R)^*$) as shown in figure.

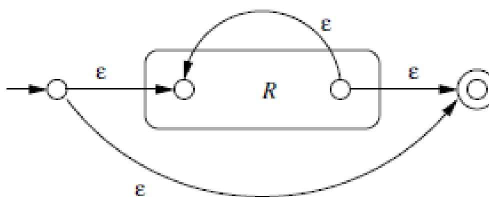


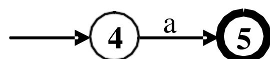
Fig. To accept the language $L(R)^*$

It is clear from figure that the machine can either accept ε or any number of $L(R_1)$s thus accepting the language $L(R_1)^*$. Here, q_0 is the start state q_f is the final state.

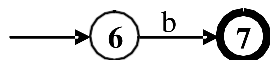
Example: Obtain an NFA which accepts strings of a's and b's starting with the string ab.

The regular expression corresponding to this language is ab(a+b)*.

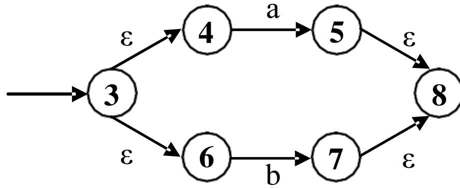
Step 1: The machine to accept 'a' is shown below.



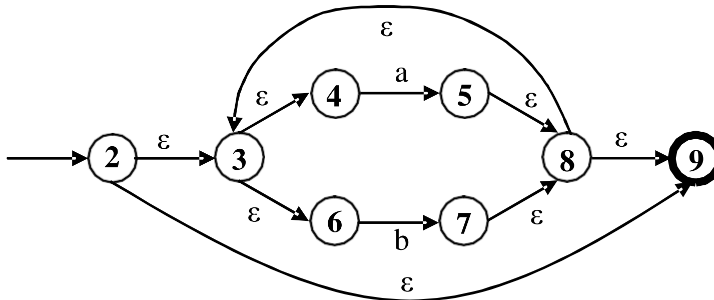
Step 2: The machine to accept 'b' is shown below.



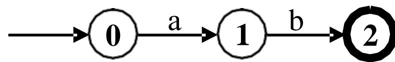
Step 3: The machine to accept (a + b) is shown below.



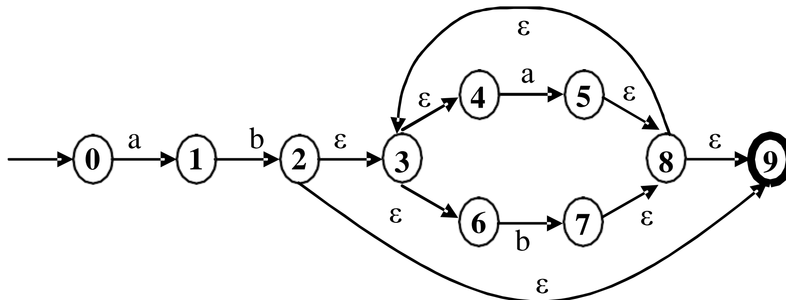
Step 4: The machine to accept (a+b)* is shown below.



Step 5: The machine to accept ab is shown below.



Step 6: The machine to accept ab(a+b)* is shown below.



To accept the language L(ab(a+b)*)

Obtain the regular expression from FA

Theorem: Let M = (Q, ∑, δ, q_0, A) be an FA recognizing the language L. Then there exists an equivalent regular expression R for the regular language L such that L = L(R).

The general procedure to obtain a regular expression from FA is shown below. Consider the generalized graph

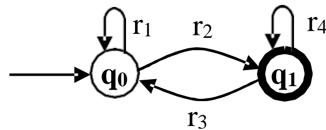


Fig. Generalized transition graph

where r_1, r_2, r_3 and r_4 are the regular expressions and correspond to the labels for the edges.The regular expression for this can take the form:

$$r=r_1^*r_2(r_3r_1^*r_2+r_4)^* \quad (3.1)$$

Note:

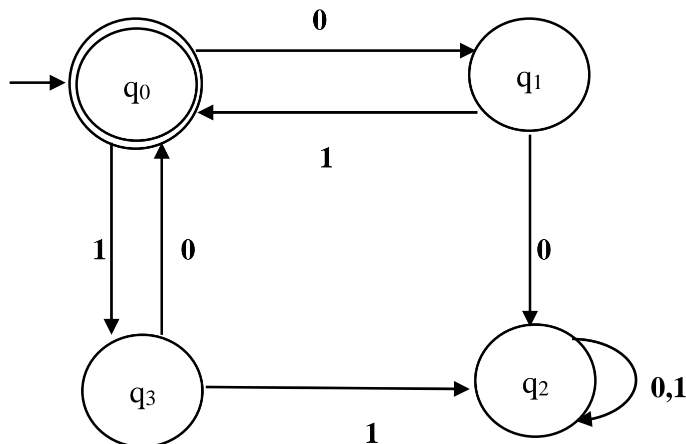
1. Any graph can be reduced to the graph shown in figure. Then substitute the regular expressions appropriately in the equation 3.1 and obtain the final regular expression.
2. If r_3 is not there in figure, the regular expression can be of the form

$$r = r_1^*r_2r_4^* \quad (3.2)$$

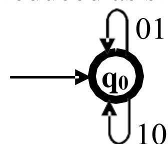
3. If q_0 and q_1 are the final states then the regular expression can be of the form

$$r = r_1^* + r_1^*r_2r_4^* \quad (3.3)$$

Example: Obtain a regular expression for the FA shown below:



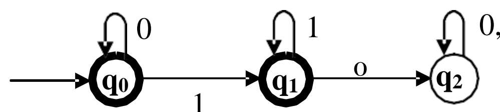
The figure can be reduced as shown below:



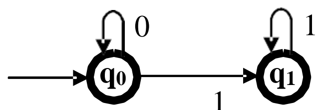
It is clear from this figure that the machine accepts strings of 01's and 10's of any length and the regular expression can be of the form

$$(01 + 10)^*$$

Example: What is the language accepted by the following FA



Since, state q_2 is the dead state, it can be removed and the following FA is obtained.



The state q_0 is the final state and at this point it can accept any number of 0's which can be represented using notation as

$$0^*$$

q_1 is also the final state. So, to reach q_1 one can input any number of 0's followed by 1 and followed by any number of 1's and can be represented as

$$0^*11^*$$

So, the final regular expression is obtained by adding 0^* and 0^*11^*. So, the regular expression is

$$\begin{aligned} R.E &= 0^* + 0^*11^* \\ &= 0^*(\ \varepsilon + 11^*) \\ &= 0^*(\ \varepsilon + 1^+) \\ &= 0^*(1^*) = 0^*1^* \end{aligned}$$

It is clear from the regular expression that language consists of any number of 0's (possibly ε) followed by any number of 1's(possibly ε).

THEOREM 6.2 For Every FSM There is an Equivalent Regular Expression

Theorem: Every regular language (i.e. every language that can be accepted by some FSM) can he defined with a regular expression.

Proof: the proof is by construction. Given an FSM $M = (K.\ \sum,\ \delta.\ s.\ A)$, we can construct a regular expression α such that L (M) = L (α).

As we did in fsmtoregexheuristic we will begin by assuring that *M* has no unreachahle states and that it has a start state that has no transitions into it and a single accepting state that has no transitions out from it. But now we will make a further important modification to *M* before we start removing states: From

every state other than the accepting state there must be exactly one transition to everystate (including itself) exceptthe start state. And into every state other than the start state there must be exactly one transition from every state (including itself) except the accepting state. to make this true. we do two things:

• If there is more than one transition between states *p* and *q,* collapse them into a single transition. If the set of labels on the original set of such transitions is

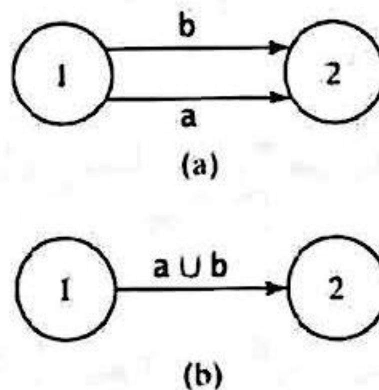
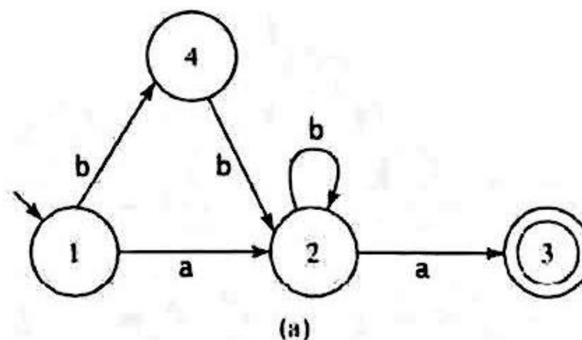


FIGURE: Collapsing multiplc transitions into one ..

{ c1, c2, ••• , c, }, then delete those transitions and replace them by a single transition with the label c1 U c2 U ... U c_n. For example. consider the FSM fragment shown in Figure (a). We must collapse the two transitions between states 1 and 2. After doing so. we have the fragment shown in Figure (b).

If any of the required transitions are missing. add them. We can add all of those transitions without changing *L(M)* by labeling all of the new transitions with the regular expression Ø. So there is no string that will allow them to be taken. For example. let *M* be the FSM shown in Figure (a). Several new transitions are required. When we add them, we have the new FSM shown in Figure (b).



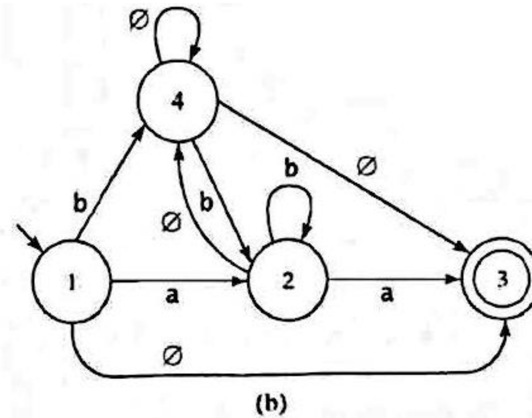


FIGURE. Adding all the required transitions.

Now suppose that we select a state *rip* and remove it and the transitions into and out of it. Then we must modify every remaining transition so that M's function stays the same. Since *M* already contains a transition between each pair of states (except the ones that are not allowed into and out of the start and accepting states), if all those transitions are modified correctly then M's behavior will be correct.

So, suppose that we remove some state that we will call *rip.* How should the remaining transitions be changed? Consider any pair of states *p* and *q.* Once were move *rip,* how can *M* get from *p* to *q?*

• lt can still take the transition that went directly from *p* to *q,* or

• It can take the transition from *p* to *rip.* Then, it can take the transition from *rip* back to itself zero or more times. Then it can take the transition from *rip* to *q.*

Let *R(p,* q) be the regular expression that labels the transition in *M* from *p* to *q.* Then, in the new machine *M'* that will be created by removing *rip,* the new regular expression that should label the transition from *p* to *q* is:

R(p, q) /* Go directly fromp to

U /*or

R(p, rip) /* go from *p* to *rip,* then

*R(rip, rip)** /* l•go from *rip* back to itself any number of times. then

R(rip, q) /* *I** go from *rip* to *q.*

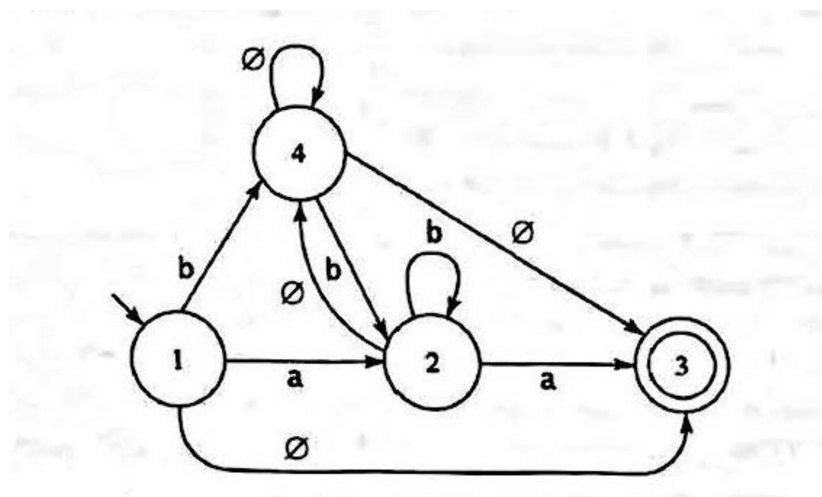
We'll denote this new regular expression *R* '(p, q). Writing it out without the comments1 we have:

***R' = R(p,* q) U *R(p, rip)R(rip, rip)*R(rip,* q).**

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}(R_{11}^{(0)})^* R_{1j}^{(0)}$$

EXAMPLE: Ripping States Out One at a Time

Again. let *M* be:



let *rip* be state 2. Then:

R'(1 , 3) = *R(1,* 3) U R(1, *rip)R(rip, rip)*R(rip,* 3).

= R(l, 3) U R(l, 2)R(2, 2)*R(2, 3).

= Ø U a b* a.

= ab*a.

Notice that ripping state 2 also changes another way the original machine had to get from state 1 to state 3: It could have gone from state 1 to state 4 to state 2 and then to state 3. But we don't have to worry about that in computing *R'* (1, 3). The required change to that path will occur when we compute R' (4, 3).

When all states except the start state s and the accepting state *a* have been removed, *R(s, a)* will describe the set of strings that can drive *M* from its start state to its accepting state. So *R(s,* a) will describe L(M).

We can now define an algorithm to build, from any FSM $M = (K. \sum, \delta, s, A)$, a regular expression that describes *L(M).* We' ll use two suhroutines, standardize, which will convert *M* to the required form. And *buildregex,* which will construct, from the modified machine *M.* the required regular expression.

standardize *(M:* FSM) =

1. Remove from *M* any states that are unreachable from the start state.
2. If the start state of *M* is part of a loop (i.e., it has any transitions coming into it). create a new start state *s* and connect *s* to M's start state via an ε-transition.
3. If there is more than one accepting state of *M* or if then.• is just one but there are any transitions out of it, create a new accepting state and connect each of M's accepting states to it via an ε-transition. Remove the old accepting states from the set of accepting states.
4. If there is more than one transition between states *p* and *q.* collapse them into a single transition.
5. If there is a pair of states *p. q* and there is no transition between them and *p* is not the accepting state and *q* is not the start state, then create a transition from *p* to *q* labelled Ø .

***buildregex(M:* FSM) =**

1. If *M* has no accepting states, then halt and return the simple regular expression Ø.

2. If *M* has only one stale, then halt and return the simple regular expression *ε.*

3. Until only the start slate and the accepting state remain do:

3.1. Select some state *rip* of *M.* Any state except the start state or the accepting state may be chosen

3.2. For every transition from some state *p* to some state *q,* if both *p* and *q* are not *rip* then using the current labels given by the expressions *R,* computethe new label *R* ' for the transition from *p* to *q* using the formula:

$$R'(p, q) = R(p, q) \cup R(p, rip)R(rip, rip)^*R(rip, q).$$

3.3. Remove *rip* and all transitions into and out of it.

4. Return the regular expression that labels the one remaining transition from the start state to the accepting state.

We can show that the new FSM that is built by standardize is equivalent to the original machine (i.e., that they accept the same language) by showing that the language that is accepted is preserved at each step of the procedure. We can show that*builtiregex(M)* builds a regular expression that correctly defines L(M) by induction on the number of states that must be removed before it halts. Using those two

procedures, we can now define:

fsmtoregex(M: **FSM) =**

1. ***M'*** = ***standardize*** **(M).**

2. Return ***buildregex(M').***

Regular languages

In theoretical computer science and formal language theory, a **regular language** is a formal language that can be expressed using a regular expression. Note that the "regular expression"features provided with many programming languages are augmented with features that makethem capable of recognizing languages that cannot be expressed by the formal regular expressions (*as formally defined below*).

In the Chomsky hierarchy, regular languages are defined to be the languages that are generated by Type-3 grammars (regular grammars). Regular languages are very useful in input parsing and programming language design.

Formal definition

The collection of regular languages over an alphabet Σ is defined recursively as follows:

- The empty language Ø is a regular language.
- For each $a \in \Sigma$ (a belongs to Σ), the singleton language $\{a\}$ is a regular language.
- If A and B are regular languages, then $A \cup B$ (union), $A \bullet B$ (concatenation), and A^*(Kleene star) are regular languages.
- No other languages over Σ are regular.

Examples:

All finite languages are regular; in particular the empty string language $\{\varepsilon\} = Ø^*$ is regular. Other typical examples include the language consisting of all strings over the alphabet $\{a, b\}$which contain an even number of as, or the language consisting of all strings of the form: several as followed by several bs.

A simple example of a language that is not regular is the set of strings $\{a^n b^n | n>=0\}$ Intuitively, it cannot be recognized with a finite automaton, since a finite automaton has finite memory and it cannot remember the exact number of a's. Techniques to prove this factrigorously are given below.

Proving languages not to be regular languages

We have established that the class of languages known as the regular languages has at least four different descriptions. They are the languages accepted by DFA's by NFA's and by ε - NFA's they are also the languages defined by regular expressions.

Not every language is a regular language, Here, we shall introduce a powerful technique, known as the "pumping lemma", for showing certain languages not to be regular.

- Pumping Lemma

  Used to prove certain languages like $L = \{0^n 1^n \mid n \geq 1\}$ are not regular.

- Closure properties of regular languages

  Used to build recognizers for languages that are constructed from other languages bycertain operations.

  Ex. Automata for intersection of two regular languages

- Decision properties of regular languages

  – Used to find whether two automata define the same language

  – Used to minimize the states of DFA

  eg. Design of switching circuits.

Let $L = \{0^n 1^n \mid n \geq 1\}$

There is no regular expression to define L. 00*11* is not the regular expression defining L.

Pumping Lemma for regular languages (Explanation)

Pumping Lemma (PL) for Regular Languages

Theorem:

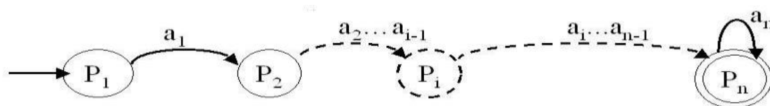
Let L be a regular language. Then there exists a constant 'n' (which depends on L) such that for every string w in L such that $|w| \geq n$, we can break w into three strings,w=xyz, such that:

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

PROOF:

Let L be regular defined by an FA having 'n' states. Let w= a_1, a_2, a_3 -- a_n and is in L.

$|w| = n \geq n$. Let the start state be P_1. Let w = xyz where x= $a_1, a_2, a_3, \ldots, a_{n-1}$, y=a_n and z = ε.



$\delta(P_1,a_1)=P_2$

$\delta(P_2,a_2)=P_3$

⋮

$\delta(P_n,a_n)=P_{n+1}$

But there are only n states. => there must be a loop. Let there be a loop in P_n State.

Let $x=a_1,\ldots\ldots a_{n-1}$

$y=a_n$

$z=\varepsilon$

Therefore:

$xy^kz = a_1 \text{-------} a_{n-1} (a_n)^k \varepsilon$

k=0 a_1 ------ a_{n-1} is accepted

k=1 a_1 ------ a_n is accepted

k=2 a_1 ------ a_{n+1} is accepted

k=10 a_1 ------ a_{n+9} is accepted and so on.

Uses of Pumping Lemma: - This is to be used to show that, certain languages are not regular. It should never be used to show that some language is regular. If you want to show that language is regular, write separate expression, DFA or NFA.

General Method of proof: -

1. Select w such that $|w| \geq n$
2. Select y such that $|y| \geq 1$
3. Select x such that $|xy| \leq n$
4. Assign remaining string to z
5. Select k suitably to show that, resulting string is not in L.

Example 1.

To prove that $L=\{w|w \ \varepsilon \ a^nb^n, \text{ where } n \geq 1\}$ is not regular

Proof:

Let L be regular. Let n is the constant (PL Definition). Consider a word w in L. Let $w = a^nb^n$, such that $|w|=2n$. Since $2n > n$ and L is regular it must satisfy PL.

Consider w= aa-----a bb-----b (first n symbols are a's, next n symbols are b's; xy covers the a's, z covers the b's)

xy contain only a's. (Because **|xy| ≤ n**).Let

|y|=l, where l > 0 (Because **|y| > 0**).

Then, the breakup of x. y and z can be as follows

$$w = \underbrace{a^{n-l}}_{x}\ \underbrace{a^{l}}_{y}\ \underbrace{b^{n}}_{z}$$

from the definition of PL , $w=xy^kz$, where k=0,1,2, ------------ ∞, should belong to L.

That is $a^{n-l}\ (a^l)^k\ b^n \in L$, for all k=0,1,2,…..,∞

Put k=0. we get $a^{n-l}\ b^n \in L$.

Contradiction. Hence the Language is not regular.

Example 2.

To prove that L={w|w is a palindrome on {a,b}*} is not regular. i.e., L={aabaa, aba,abbbba,…}

Proof:

Let L be regular. Let n is the constant (PL Definition). Consider a word w in L. Let w= a^nba^n, such that |w|=2n+1.

Since 2n+1 > n and L is regular it must satisfy PL.

Consider w= aa-----a b aa-----a (first n symbols are a's, then b, then n a's; xy covers the first a's, z covers the rest)

xy contain only a's. **(**Because **|xy| ≤ n).**

Let |y|=l, where l > 0 **(**Because **|y| > 0).**

That is, the break up of x. y and z can be as follows

$$w = \underbrace{a^{n-l}}_{x}\ \underbrace{a^{l}}_{y}\ \underbrace{ba^{n}}_{z}$$

from **the definition** of PL $w=xy^kz$, where k=0,1,2,-------------∞, should belong to L.

That is $a^{n-l}(a^l)^k b(a)^n \in L$, for all k=0,1,2,…,∞.

Put k=0. we get a^{n-1} b a^n $\notin$L, because, it is not a palindrome. Contradiction, hence the language is not regular

Example 3.

To prove that L={ all strings of 1's whose length is prime} is not regular. i.e., L={1^2,1^3 ,1^5 ,1^7 ,1^{11} , ------------ }

Proof: Let L be regular. Let w = 1^p where p is prime and | p| = n +2

Let y = m.

by PL $xy^kz \in L$

$| xy^kz | = | xz | + | y^k |$

Let k = p-m

= (p-m) + m (p-m)

= (p-m)(1+m) -----this can not be prime if p-m $\geq$ 2 or 1+m $\geq$ 2

1. (1+m) $\geq$ 2 because m $\geq$ 1
2. Limiting case p=n+2

   (p-m) $\geq$ 2 since m $\leq$n

Example 4.

To prove that L={ 0^{i^2} | i is integer and i >0} is not regular. i.e., L={0^2 , 0^4 ,0^9 ,0^{16} ,0^{25} ,...}

Proof: Let L be regular. Let w = 0^{n^2} where |w| = $n^2 \geq n$

by PL $xy^kz \in$ L, for all k = 0,1,---

Select k = 2

$| xy^2z | = | xyz | + | y |$

= n^2 + Min 1 and Max n

Therefore $n^2 < | xy^2z | \leq n^2 + n$

$n^2 < | xy^2z | < n^2 + n + 1+n$ adding 1 + n (Note that less than or equal to is

$n^2 < | xy^2z | < (n + 1)^2$ replaced by less than sign)

Say n = 5 this implies that string can have length > 25 and < 36 which is not of the form 0^{i2}.

Lexical analysis

Lexical analysis reads characters from left to right and groups into tokens. A simple way to build lexical analyzer is to construct a diagram to illustrate the structure of tokens of the source program. We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a *lexical-analyzer generator* and compiling those patterns into code that functions as a lexical analyzer. This approach makes it easier to modify a lexical analyzer, since we have only to rewrite the affected patterns, not the entire program.

Three general approaches for implementing lexical analyzer are:

i. Use lexical analyzer generator (LEX) from a regular expression based specification that provides routines for reading and buffering the input.

ii. Write lexical analyzer in conventional language using I/O facilities to read input.

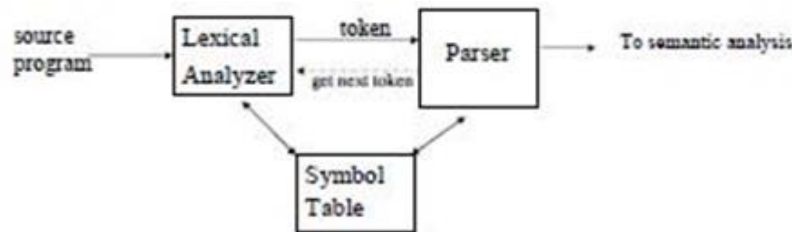
iii. Write lexical analyzer in assembly language and explicitly manage the reading of input.

The speed of lexical analysis is a concern in compiler design, since only this phase reads the source program character-by character.

The role of lexical analyzer

The lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and *whitespace* (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

It is the first phase of a compiler. It reads source code as input and sequence of tokens as output. This will be used as input by the parser in syntax analysis. Upon receiving 'getNextToken' from parser, lexical analyzer searches for the next token.



Some additional tasks are: eliminating comments, blanks, tab and newline characters, providing line numbers associated with error messages and making a copy of the source program with error messages. Some of the issues are: simpler design, compiler efficiency is improved and compiler portability is enhanced.

Tokens, Patterns, and Lexemes

Token: Token is a terminal symbol in the grammar for the source language. When the character sequence 'pi' appears in the source program, a token representing identifier is returned to the parser. A *token* is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

Pattern: Pattern is a rule describing the set of lexemes that can represent a particular token in source programs. A *pattern* is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.

Lexeme: Lexeme is a sequence of characters in the source program that is matched by the pattern for a token. A *lexeme* is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Examples of tokens

Token	Informal description	Sample Lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
Comparison	< or> or <= or >= or == or !=	<=, !=
id	Letter followed by letters and digits	Pi, score, D2
Number	Any numeric constant	3.14, 0.6, 20
Literal	Anything but surrounded by " " 's	"total= %d\n" , "core dumped"

In many programming languages, the following classes cover most or all of the tokens:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token comparison.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Case Study1: List out lexeme and token in the following example

```
int Max(int a, int b)
{ if(a>b)
    return a;
  else
    return b;
}
```

Lexeme	Token
int	Keyword
Max	Identifier
(	Operator
a	Identifier
,	Operator
.	.
.	.
.	.

Attributes for tokens

A token has only a single attribute – a pointer to the symbol-table entry in which the information about the token is kept. The token names and associated attribute values for the statement E = M * C ** 2 are written below as a sequence of pairs. <id, pointer to symbol-table entry for E>

<assign_op>

<id, pointer to symbol-table entry for M>

<mult_op>

<id, pointer to symbol-table entry for C>

<exp_op>

<number, integer value 2>

Lexical errors

It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For instance, if the string f i is encountered for the first time in a C program in the context:

f i (a == f (x)) . ..

A lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier. Since fi is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.

However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. The simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate. Other possible error-recovery actions are:

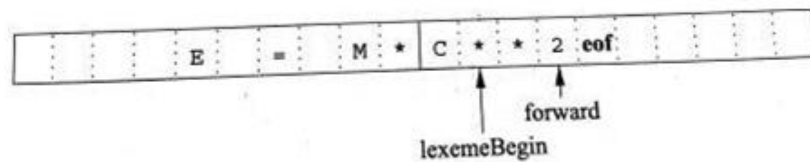
1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

Input buffering

Input buffering is an important task of reading the source program to speedup. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.

Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. Look ahead of characters in the input is necessary to identify tokens. Specified buffering techniques have been developed to reduce the large amount of time consumed in moving characters. A buffer (array) divided into two N-character halves, where N=number of characters on one disk block 'eof' marks the end of source file and it is different from input character.



Using a pair of input buffers

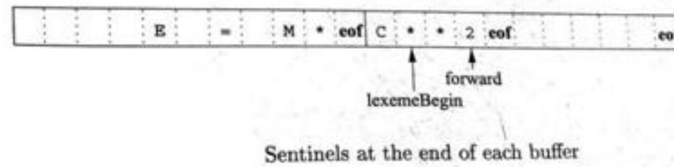
Two pointers are maintained: beginning of the lexeme pointer and forward pointer.

Initially, both pointers point to the first character of the next lexeme to be found. Forward pointer scans ahead until a match for a pattern is found. Once the next lexeme is determined, processed and both pointers are set to the character immediately past the lexeme. If the forward pointer moves halfway mark, the right N half is filled with new characters. If the forward pointer moves right end of the buffer then left N half is filled with new characters. The disadvantage is look ahead is limited and it is impossible to recognize tokens, when distance between the two pointers is more than the length of the buffer.

Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than *N,* we shall never overwrite the lexeme in its buffer before determining it.

Sentinels

Instead of this, we provide an extra character, sentinel at the end of each half of the buffer. Sentinel is not part of our source program and works as 'eof'. Now only one test is sufficient, that is, if forward='eof' or not. It is an extra key inserted at the end of the array. It is a special, dummy character that can't be part of source program.



With respect to buffer pairs, the code for advancing forward pointer is:

Lookahead code with sentinels

```
Switch(*forward++)
 {
 case eof:
          if (forward is at end of first buffer)
          { reload second buffer;
            forward = beginning of second buffer;
          }
    else if (forward is at end of second buffer)
          { reload first buffer;
            forward = beginning of first buffer;
          }
    else /* eof within a buffer marks the end of input terminate lexical analysis*/
    break;
     case for the other characters
 }
```

Specifications of token

An *alphabet* is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set {0,1} is the *binary alphabet.*

A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms "sentence" and "word" are often used as synonyms for "string."

The length of a string *s,* usually written |s|, is the number of occurrences of symbols in *s.* For example, banana is a string of length six. The *empty string,* denoted €, is the string of length zero.

The following string-related terms are commonly used:

1. A *prefix* of string *s* is any string obtained by removing zero or more. symbols from the end of *s.* For example, ban, banana, and e are prefixes of banana.
2. A *suffix* of string *s* is any string obtained by removing zero or more symbols from the beginning of *s.* For example, nana, banana, and e are suffixes of banana.
3. A *substring* of *s* is obtained by deleting any prefix and any suffix from *s.* For instance, banana, nan, and e are substrings of banana.
4. The *proper* prefixes, suffixes, and substrings of a string *s* are those, prefixes, suffixes, and substrings, respectively, of *s* that are not e or not equal to *s* itself.
5. A *subsequence* of *s* is any string formed by deleting zero or more not necessarily consecutive positions of *s.* For example, baan is a subsequence of banana.

A *language* is any countable set of strings over some fixed alphabet. This definition is very broad. Abstract languages like 0, the *empty set,* or {ε}, the set containing only the empty string, are languages under this definition. So too are the set of all syntactically well-formed C programs and the set of all grammatically correct English sentences, although the latter two languages are difficult to specify exactly.

Operations on languages

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Definitions of operations on languages

Case Study2:

Let L be the set of letters {A, B, ..., Z, a, b, ..., z} and let D be the set of digits {0, 1, ... 9}. We may think of L and D in two, essentially equivalent, ways. One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages L and D, using the operators of Fig. 3.6:

1. $L \cup D$ is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3. L^4 is the set of all 4-letter strings.
4. L^* is the set of all strings of letters, including ε, the empty string.
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

Regular expressions

It is a notation which allows defining the sets precisely. eg: L(LUD)* its regular expression is: letter(letter|digit)*

Regular expression over alphabet has following rules:

- ε is a regular expression, the set containing empty string is a regular expression.
- a is a regular expression.
- Suppose r and s are regular expression denoting the languages L(r) and L(s) then, (r|s) is a regular expression denoting L(r)UL(s),

rs is a regular expression denoting L(r)L(s),

(r)* is a regular expression denoting (L(r))*,

(r) is a regular expression denoting L(r),

A language denoted by regular expression is said to be a regular set. *, concatenation and | has highest to lowest precedence with left associative. If two regular expression ‘r’ and ‘s’ denote the same language, we say ‘r’ and ‘s’ are equivalent and write r=s.

BASIS: There are two rules that form the basis:

1. ε is a regular expression, and ***L( )* is {ε}**, that is, the language whose sole member is the empty string.

2. If *a* is a symbol in E, then a is a regular expression, and L(a) = *{a},* that is, the language with one string, of length one, with *a* in its one position.

Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and *s* are regular expressions denoting languages *L(r)* and *L(s),* respectively.

1. (r)|(s) is a regular expression denoting the language *L(r)* U *L(s).*

2. (r)(s) is a regular expression denoting the language *L(r)L(s).*

3. (r)* is a regular expression denoting (L(r))*.

4. (r) is a regular expression denoting *L(r).* This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote. The following table shows some of the regular expressions along with the possible regular sets:

Regular expression set a|b {a,b}

(a|b)(a|b) or aa|ab|ba|bb {aa,ab,ba,bb}

a* {€,a,aa,aaa,…}

(a|b)* or (a*b*)* {€, a,aa,b,bb,…}

a|a*b {a,b,ab,aab,aaab,…}

Algebraic properties (laws) of regular expression

LAW	DESCRIPTION
$r \mid s = s \mid r$	$\mid$ is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s \mid t) = rs \mid rt;\ (s \mid t)r = sr \mid tr$	Concatenation distributes over $\mid$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r \mid \epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

Regular definitions

It is the process of giving a name to certain regular expressions and use those names in subsequent expressions. If ε is an alphabet of basic symbols, then regular definition is a sequence of definitions of the form,

d1→r1, d2 →r2 and so on where di is distinct name and ri is regular expression. For example,

- A *regular definition* is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and
$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in
. $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$
$d_n \rightarrow r_n$

basic symbols previously defined names

Case study 3:

1) Regular definition for c identifier

2) Regular definition for unsigned number

1) Regular definition for C identifier is given by,

letter → A|B|..|Z|a|b|..|z

digit→ 0|1|..|9

id→ letter(letter|digit)*

2) Regular definition for unsigned number is given by,

digit→ 0|1|..|9

Digits → digit digit*

OptionalFraction → . Digits | €

OptionalExponent → (E(+|-|€)Digits)| €

Number→ Digits OptionalFraction OptionalExponent

Extensions of regular expressions (Notational shorthands)

One or more instances: unary postfix operator '+' eg: if 'r' is a regular expression denoting the language L(r) then (r)+ is a regular expression denoting the language (L(r))+

a+ is a regular expression of set of all strings of one or more a's r* = r+ | **ε**

r+ = rr*

Zero or one instance :unary postfix operator '?'

eg: r? means r|**ε**

If 'r' is a regular expression denoting the language L(r) then (r)? is a regular expression denoting the language L(r)U{**ε**}

Character class

[abc] denotes the regular expression a|b|c

[a-z] denotes the regular expression a|b|..|z

The above 2 regular definitions are given by,

1) Letter→ [a-zA-Z_]

   Digit → [0-9]

   Id → Letter_(Letter_|Digit)*

2) Digit → [0-9]

   Digits → Digit+

   Number → Digits(. Digits)?(E[+ -]?Digits)?

Recognition of tokens

Recognition of tokens take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns. Consider the following grammar, fragment/regular definitions:

Stmt→ if expr then stmt | if expr then stmt else stmt | **ε**

Expr→term relop term | term

Term→ id | num

if → if

then → then

else → else

relop → <|<=|>|>=|=|<>

id → letter(letter|digit)*

num → digit+(.digit+)?(_(+|-)?digit+)?

- **Keywords cannot be used as identifiers**
- **Num represents unsigned int and real numbers**
- **The regular definition for blank(white space) is,**

delim → blank | tab | newline

ws→ delim+

If 'ws' is found, the lexical analyzer does not return a token to the parser. Our goal is to construct a lexical analyzer to produce a pair consisting of token and attribute-value as output using the translation table.

Lexeme	Token	Attribute
Whitespace	ws	—
if	if	—
then	then	—
else	else	—
An identifier	id	Pointer to table entry
A number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Transition diagrams.

These are the flow charts, as an intermediate step in the construction of a lexical analyzer. This takes actions when a lexical analyzer is called by the parser to get the next token. We use transition diagram to keep track of information about characters that are seen as and when the forward pointer scans the input. Lexeme beginning pointer points to the character following the last lexeme found. E=M* C**2eof

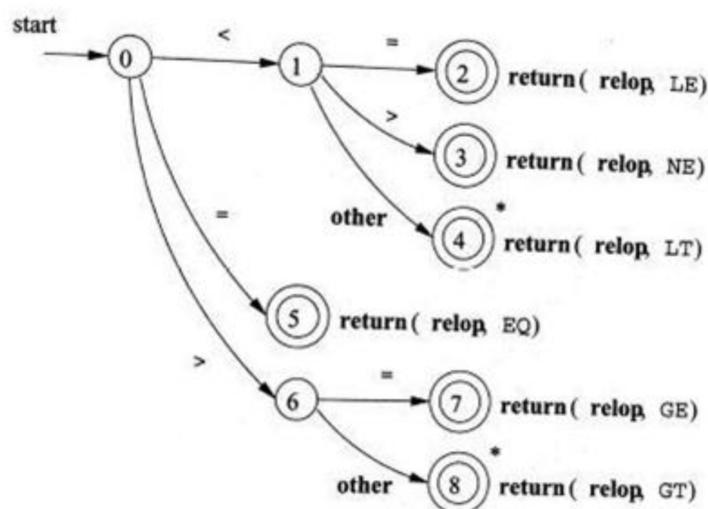
Transition diagrams have a collection of nodes or circles, called *states.* Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of a state as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the *forward* pointer.

Edges are directed from one state of the transition diagram to another. Each edge is *labeled* by a symbol or set of symbols. If we are in some state, and the next input symbol is *a,* we look for an

edge out of state *s* labeled by *a* (and perhaps by other symbols, as well). If we find such an edge, we advance the *forward* pointer arid enter the state of the transition diagram to which that edge leads. We shall assume that all our transition diagrams are *deterministic,* meaning that there is never more than one edge out of a given state with a given symbol among its labels. We shall relax the condition of determinism, making life much easier for the designer of a lexical analyzer, although trickier for the implementer. Some important conventions about transition diagrams are:

1. Certain states are said to be *accepting,* or *final.* These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the *lexemeBegin* and *forward* pointers. We always indicate an accepting state by a double circle, and if there is an action to be taken
— typically returning a token and an attribute value to the parser — we shall attach that action to the accepting state.
2. In addition, if it is necessary to retract the *forward* pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract *forward* by more than one position, but if it were, we could attach any number of *'s to the accepting state.
3. One state is designated the *start state,* or *initial state;* it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Case study 4: Construct the Transition diagram for *relop*



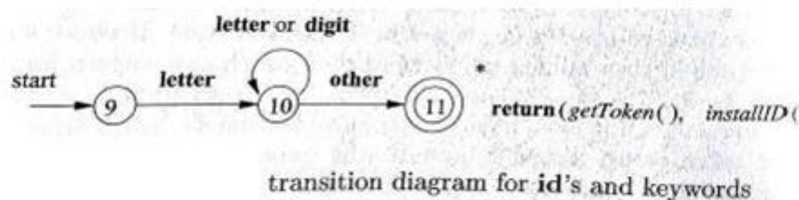
Transition diagram for **relop**

Case study 5: Construct the Transition diagram for Recognition of Reserved Words and Identifiers:

There are two ways that we can handle reserved words that look like identifiers:

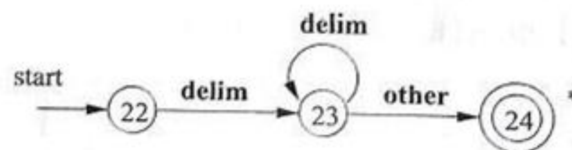
1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. When we find an identifier, a call to *installlD* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is **id.** The function *getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either **id** or one of the keyword tokens that was initially installed in the table.

2. Create separate transition diagrams for each keyword; an example for the keyword **then**. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token **then** in situations where the correct token was **id,** with a lexeme like thennextvalue that has **then** as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved word tokens are recognized in preference to id, when the lexeme matches both patterns.

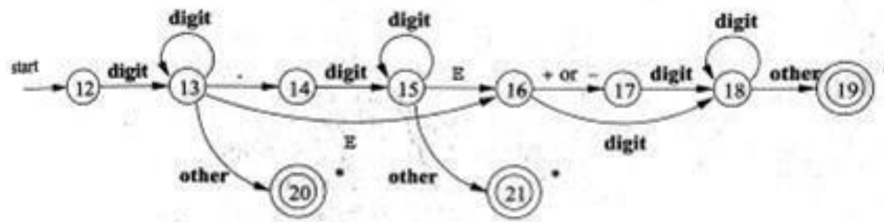


transition diagram for id's and keywords

Transition diagram for whitespace:



Transition diagram for unsigned numbers:



A transition diagram for unsigned numbers

Implementation of *relop* transition diagram:

```
TOKEN getRelopO
{
TOKEN retToken = new(RELOP);
while(1) { /* repeat character processing until a return or failure occurs */
switch(state) {
case 0:
           c = nextChar( );
           if ( c == ‘<’ ) state = 1;
          else if ( c == '=' ) state = 5;
          else if ( c == '>' ) state = 6;
          else fail(); /* lexeme is not a relop */
           break;
case 1: ......

case 2: ….
case 8: retract();
           retToken.attribute = GT;
return(retToken); }
}
```

A sequence of transition diagrams can be converted into a program to look for the tokens. The size of the program is propositional to the number of states and edges in the diagrams. Each state gets a segment of code (module). Nextchar() is to read a next char from the input buffer. fail() calls an error recovery routine, when error is encountered. lexical_value returns tokens when id or num is found by install-id() and install_num() respectively. State & start keep track of the present and start state. There are several ways that a collection of transition diagrams can be used to build a lexical analyzer. Regardless of the overall strategy, each state is represented by a piece of code. We may imagine a variable state holding the number of the current state for a transition diagram. A switch based on the value of state takes us to code for each of the possible states, where we find the action of that state. Often, the code for a state is itself a switch statement or multiway branch that determines the next state by reading and examining the next input character.

Question Bank

1.Give regular expression for the following languages:

a) L= {W/W is in {a,b}* and |W| mod 3= 0}

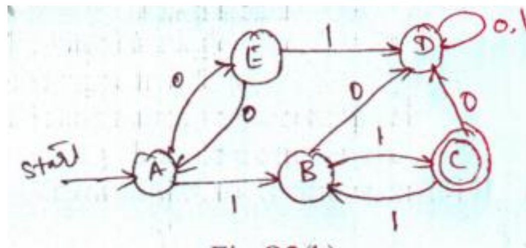
b) L = {W/W is a string of even number of 0's followed by odd number of 1's}.

2. Define regular expression and also write the regular expression for the following languages:

a)L= {w€ {a,b}* |w has exactly one pair of consecutive a's}.

b)Set of all strings not ending in substring 'ab' over ∑ = {a,b}.

3. Define Regular expression . Convert the following automation to a regular expression using state elimination technique.



4. Convert the regular expression (0+1)* | (0+1) to an NFA.

5. Prove that following language is not regular using pumping theorem L = {$a^n b^n$ | n >=0}.

6. Prove that following language is not regular using pumping theorem L = {ww^r | w ∈{a,b}*}.

7. Construct the NFA for the regular expression (a+b)*aa(a+b)*.

8. Develop the regular expressions for ∑={a,b}:

i) not more than 3 a's

ii) L={$a^n b^m$|n>=4 m<=3}

iii) L={w : $n_a(w)$ mod 3=0}

9: Define Lexical Analysis. List the approaches for implementing lexical analyser.

10. Explain the role of the lexical analyser in building a compiler.

OR

Explain the use and coordination between 'LEX' AND 'YACC' the compiler writing tools.

11: Define the following

i. Tokens

ii. Patterns

iii. Lexemes

12: Explain attribute values for tokens

13: List the commonly occurring Lexical errors. Explain the possible error-recovery actions.

14: Why buffering is required while recognizing lexemes? Explain how sentinels are handled

using buffers.

15: Explain specification of tokens in building a language.

16: Enlist the operations on languages

17: What is a regular expression? List the rules in building a regular expression.

18: Enlist algebraic properties (laws) of regular expression

19. Define regular definition

20: List the extensions of regular expressions (Notational shorthands)

21. Explain how tokens are recognized.

22: What are transition diagrams? How transition diagrams are useful in token recognition.

23: Develop code for implementation of *relop* transition diagram: