Vivekananda
College of Engineering & Technology
Nehru Nagar Post, Puttur, D.K. 5742013

Lecture Notes

15CS54
Automata Theory and Computability
(CBCS Scheme)

Prepared by

Mr. Harivinod N
Dept. of Computer Science and Engineering,
Vivekananda College of Engineering and Technology, Puttur

Module-2:	**Regular**	**Expression** **Grammar** **Language**

Contents

Course website:
www.techjourney.in

1. Regular Expression

Let's now take a different approach to categorizing problems. Instead of focusing on the power of a computing device, let's look at the task that we need to perform. In particular, let's consider problems in which our goal is to match finite or repeating patterns.

- The first step of compiling a program: This step is called lexical analysis. Its job is to break the source code into meaningful units such as keywords, variables, and numbers.
- Filtering email for spam
- Searching a complex directory structure by specifying patterns that are known to occur in the file we want.

Definition: Regular expression (RE) is defined in recursive way. The regular expressions over an alphabet Σ are all and only the strings that can be obtained as follows:

1. ∅is a regular expression.
2. ε is a regular expression.
3. Every element of Σ is a regular expression.
4. If α , β are regular expressions, then so is αβ.
5. If α , β are regular expressions, then so is α∪β.
6. If α is a regular expression, then so is α*.
7. If α is a regular expression, then so is α^+.
8. If α is a regular expression, then so is (α).

So, if we let Σ = {a, b }, the following strings are regular expressions:

∅, ε. a. b. (aUb)*, abbaUε,

Semantic interpretation

- $L(\emptyset) = \emptyset$. The language that contains no strings
- $L(\varepsilon) = \{\varepsilon\}$. The language that contains just the empty string.
- $L(c)$, where $c \in \Sigma = \{c\}$. The language that contains the single one-character string c
- $L(\alpha\beta) = L(\alpha)\ L(\beta)$. Concatenation of RE is same as concatenation of Languages
- $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$. Union of RE's is same as union of the two constituent languages.
- $L(\alpha^*) = (L(\alpha))^*$. * is a Klene star operation. Defines the language that is formed by concatenating together zero or more strings drawn from $L(\alpha)$.
- $L(\alpha^+) = L(\alpha\alpha^*) = L(\alpha)\ (L(\alpha))^*$. If $L(\alpha)$ is equal to ∅, then $L(\alpha^+)$ is also equal to ∅. Otherwise $L(\alpha^+)$ is language formed by concatenating together one or more strings drawn from $L(\alpha)$.
- $L((\alpha)) = L(\alpha)$.

If the meaning of a regular expression α is the language L, then we say that α ***defines*** or ***describes*** L.

Example-1

$$\begin{aligned} L((a \cup b)^*b) &= L((a \cup b)^*)L(b) \\ &= (L((a \cup b)))^*L(b) \\ &= (L(a) \cup L(b))^*L(b) \\ &= (\{a\} \cup \{b\})^*\{b\} \\ &= \{a, b\}^*\{b\}. \end{aligned}$$

So the meaning of the regular expression (a U b)*b is the set of all strings over the alphabet {a, b} that end in b.

Example-2

$$\begin{aligned} L(((a \cup b)(a \cup b))a(a \cup b)^*) &= L(((a \cup b)(a \cup b)))L(a)\ L((a \cup b)^*) \\ &= L((a \cup b)(a \cup b))\ \{a\}\ (L((a \cup b)))^* \\ &= L((a \cup b))L((a \cup b))\ \{a\}\ \{a, b\}^* \\ &= \{a, b\}\ \{a, b\}\ \{a\}\ \{a, b\}^* \end{aligned}$$

So the meaning of the regular expression ((aU b) (aU b))a(a U b)* is:

$\{xay : x \text{ and } y \text{ are strings of a's and b's and } |x| = 2\}$.

Alternatively, it is the language that contains all strings of a's and b's such that there exists a third character and it is an a.

Example-3

Let $L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$. There are two simple regular expressions both of which define L:

$((a \cup b)(a \cup b))^*$	This one can be read as, "Go through a loop zero or more times. Each time through, choose an a or b, then choose a second character (a or b)."
$(aa \cup ab \cup ba \cup bb)^*$	This one can be read as, "Go through a loop zero or more times. Each time through, choose one of the two-character sequences."

Example-4

Let $L = \{w \in \{a, b\}^* : w \text{ contains an odd number of a's}\}$. Two equally simple regular expressions that define L are:

$b^*\ (ab^*ab^*)^*\ a\ b^*$.

$b^*\ a\ b^*\ (ab^*ab^*)^*$.

Priority

The regular expression language that we have just defined provides three operators. We will assign the following precedence order to them (from highest to lowest): 1. Kleene star. 2. concatenation, and 3. union.

So the expression {a U bb*a) will be interpreted as (a U (b(b* a))).

$(\alpha \cup \varepsilon)$	Can be read as "optional α", since the expression can be satisfied either by matching α or by matching the empty string.
(a ∪ b)*	Describes the set of all strings composed of the characters a and b. More generally, given any alphabet $\Sigma = \{c_1, c_2, \ldots, c_n\}$, the language Σ^* is described by the regular expression: $(c_1 \cup c_2 \cup \cdots \cup c_n)^*$.
a* ∪ b* ≠ (a ∪ b)*	The language on the right contains the string ab, while the language on the left does not. Every string in the language on the left contains only a's or only b's.
(ab)* ≠ a*b*	The language on the left contains the string abab, while the language on the right does not. The language on the right contains the string aaabbbb, while the language on the left does not.

Refer Class notes for more examples on RE of languages

2. Kleene's theorem

Statement: Finite state machines & regular expressions define the same class of languages; i.e. They are equivalent i.e. They are equally powerful.

To prove this, we must show:

- Theorem: Any language that can be defined with a regular expression can be accepted by some FSM and so is regular. (RE to FSM)
- Theorem: Every regular language (i.e., every language that can be accepted by some DFSM) can be defined with a regular expression. (FSM to RE)

2.1. Building FSM from Regular expression

Theorem: For Every Regular Expression There is an Equivalent FSM i.e Any language that can be defined with a regular expression can be accepted by some FSM and so is regular.

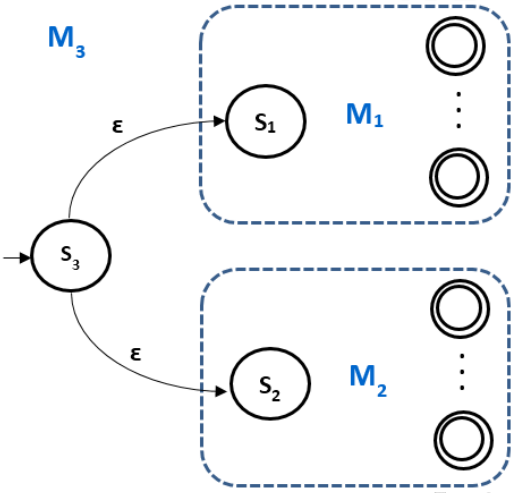
Proof: The proof is by construction. We will show that given a regular expression α, we can construct an FSM M such that L(α) = L (M).

We first show that there exists an FSM that corresponds to each primitive regular expression:

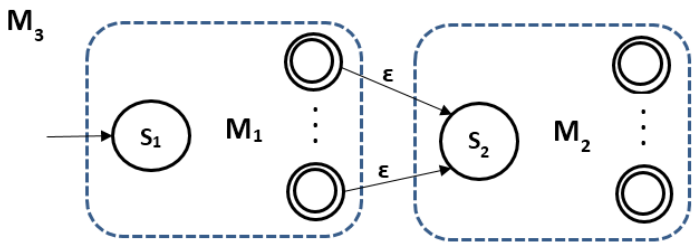
- If α is any $c \in \Sigma$, we construct for it the simple FSM shown in Figure 6.1 (a).
- If α is $\emptyset$, we construct for it the simple FSM shown in Figure 6.1 (b).
- Although it's not strictly necessary to consider ε since it has the same meaning as $\emptyset^*$ we'll do so since we don't usually think of it that way. So, if α is ε, we construct for it the simple FSM shown in Figure 6.1 (c).

Figure 6.1: FSMs for primitive regular expressions

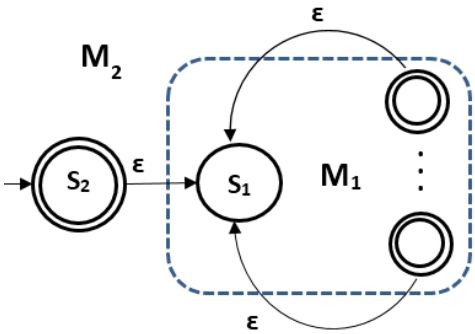
- If α is the regular expression $\beta \cup \gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, then we construct $M_3 = (K_3, \Sigma, \delta_3, s_3, A_3)$ such that $L(M_3) = L(\alpha) = L(\beta) \cup L(\gamma)$. If necessary, rename the states of M_1 and M_2 so that $K_1 \cap K_2 = \emptyset$. Create a new start state, s_3, and connect it to the start states of M_1 and M_2 via ε-transitions. M_3 accepts iff either M_1 or M_2 accepts. So $M_3 = (\{s_3\} \cup K_1 \cup K_2, \Sigma, \delta_3, s_3, A_1 \cup A_2)$, where $\delta_3 = \delta_1 \cup \delta_2 \cup \{((s_3, \varepsilon), s_1), ((s_3, \varepsilon), s_2)\}$.



- If α is the regular expression $\beta\gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, then we construct $M_3 = (K_3, \Sigma, \delta_3, s_3, A_3)$ such that $L(M_3) = L(\alpha) = L(\beta)L(\gamma)$. If necessary, rename the states of M_1 and M_2 so that $K_1 \cap K_2 = \emptyset$. We will build M_3 by connecting every accepting state of M_1 to the start state of M_2 via an ε-transition. M_3 will start in the start state of M_1 and will accept iff M_2 does. So $M_3 = (K_1 \cup K_2, \Sigma, \delta_3, s_1, A_2)$, where $\delta_3 = \delta_1 \cup \delta_2 \cup \{((q, \varepsilon), s_2) : q \in A_1\}$.



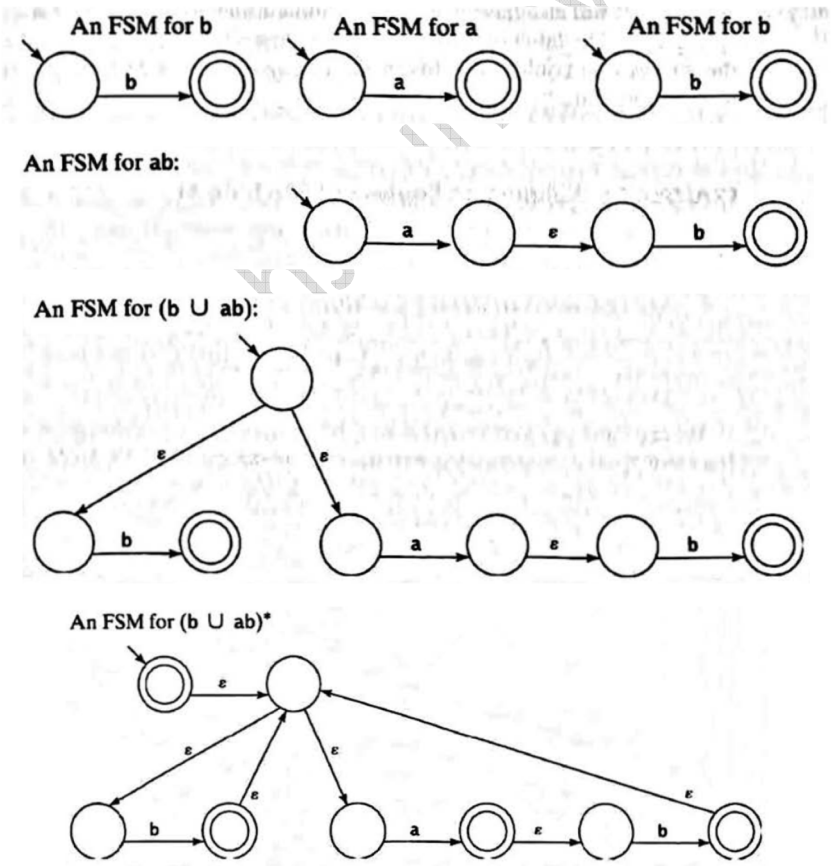
- If α is the regular expression β^* and if $L(\beta)$ is regular, then we construct $M_2 = (K_2, \Sigma, \delta_2, s_2, A_2)$ such that $L(M_2) = L(\alpha) = L(\beta)^*$. We will create a new start state s_2 and make it accepting, thus assuring that M_2 accepts ε. (We need a new start state because it is possible that s_1, the start state of M_1, is not an accepting state. If it isn't and if it is reachable via any input string other than ε, then simply making it an accepting state would cause M_2 to accept strings that are not in $(L(M_1))^*$.) We link the new s_2 to s_1 via an ε-transitions. Finally, we create ε-transitions from each of M_1's accepting states back to s_1. So $M_2 = (\{s_2\} \cup K_1, \Sigma, \delta_2, s_2, \{s_2\} \cup A_1)$, where $\delta_2 = \delta_1 \cup \{((s_2, \varepsilon), s_1)\} \cup \{((q, \varepsilon), s_1) : q \in A_1\}$.



Based on the constructions that have just been described, we can define the following algorithm to construct, given a regular expression α , a corresponding (usually nondeterministic) FSM:

regextofsm(α: regular expression) =
Beginning with the primitive subexpressions of α and working outwards until an FSM for an of *a* has been built do:
Construct an FSM as described above.

Example: Consider the regular expression (b U ab)*. We use *regextofsm* to build an FSM that accepts the language defined by this regular expression:



Refer Class notes for more examples

2.2. Building Regular expression from FSM

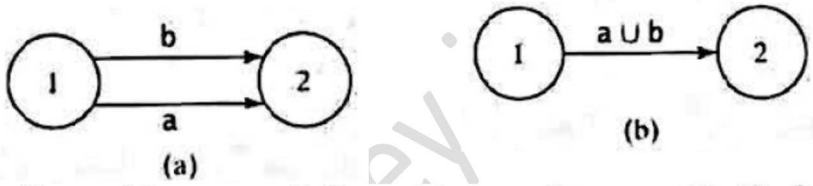
Theorem: Every regular language (i.e every language that can be accepted by some FSM) can he defined with a regular expression.

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).

Before starting the method, do the following modification.

- 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 $\{c_1, c_2, \ldots, c_n\}$, then delete those transitions and replace them by a single transition with the label $c_1 \cup c_2 \cup \ldots \cup c_n$. For example, consider the FSM fragment shown in Figure 6.2(a). We must collapse the two transitions between states 1 and 2. After doing so, we have the fragment shown in Figure 6.2(b).

Figure 6.2: Collapsing multiple transitions into one.



- 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 ∅. So there is no string that will allow them to be taken. For example, let M be the FSM shown in Figure 6.3(a). Several new transitions are required. When we add them, we have the new FSM shown in Figure 6.3(b).

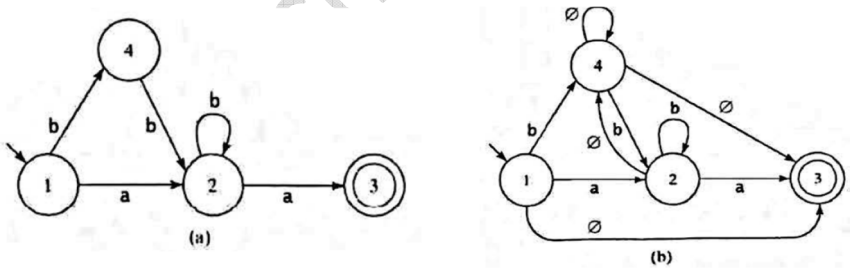


Figure 6.3. Adding all the required transitions.

Note: The above step is optional. We can assume Φ whenever it is required.

Removing a state: 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. 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 we remove *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.

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

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

We can now define an algorithm to build, from any FSM $M = (K, \Sigma, \delta, s, A)$, a regular expression that describes $L(M)$. We'll use two subroutines, *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 there 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 labeled $\emptyset$.

buildregex(*M*: FSM) =

1. If M has no accepting states, then halt and return the simple regular expression $\emptyset$.
2. If M has only one state, then halt and return the simple regular expression ε.
3. Until only the start state 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, compute the 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.

fsmtoregex(*M*: FSM) =

1. $M' = standardize\ (M)$.
2. Return *buildregex*(M').

Refer Class notes for more examples

Regular Expression, Regular Languages and Kleene's Theorem

Kleene's Theorem tells us that there is no difference between the formal power of regular expressions and finite state machines. But, as some of the examples that we just considered suggest. there is a practical difference in their effectiveness as problem solving tools:

- The regular expression language is a pattern language. In particular, regular expressions must specify the order in which a sequence of symbols must occur. This is useful when we want to describe patterns such as **phone numbers** (it matters that the area code comes first) or **email addresses** (it matters that the user name comes before the domain).
- But there are some applications where order doesn't matter: **vending machine** example that an instance of this class of problem. The order in which the coins were entered doesn't matter. **Parity checking** is another. Only the total number of 1 bits matters. not where they occur in the string. Finite state machines can he very effective in solving problems such as this. But the regular expressions that correspond to those FSMs may be too complex to be useful.

The bottom line is that sometimes it is easy to write a finite state machine to describe a language. For other problems. it may be easier to write u regular expression.

3. Applications of RE

Because patterns are everywhere, applications of regular expressions are everywhere. The term *regular expression* is used in the modern computing world in a much more general way than we have defined it here. Many programming languages and scripting systems provide support for regular expression matching. Each of them has its own syntax.

- The programming language **Perl**. for example. supports regular expression matching.
- **Decimal Numbers**: The following regular expression matches decimal encodings of numbers:

  -? ([0-9]+(\.[0-9]*)? | \.[0-9]+)

- **Biology**: Meaningful words in protein sequences arc called motifs. They can he described with regular expressions. Given a protein or DNA sequence, task is to find others that are likely to be evolutionarily close to it.

ESGHDTTTYYNKNRYPAGWNNHHDQMFFWV

To achieve this we need to, build a DFSM that can examine thousands of other sequences and find those that match any of the selected patterns.

- **lP Addresses:** The following regular expression searches for Internet (IP) addresses:

  ([0-9]{1, 3} (\. [0-9]{1, 3}){3})

- In **XML** regular expressions are one way to define parts of new document types.

- **Legal Passwords**: Consider the problem of determining whether a string is a legal password. Suppose that we require that all passwords meet the following requirements:
    - A password must begin with a letter.
    - A password may contain only letters. numbers. and the underscore character.
    - A password must contain at least four characters and no more than eight characters.

  The following regular expression describes the language of legal passwords. The line breaks have no significance. We have used them just to make the expression easier to read.

  ((a-z) U (A-Z))
  ((a-z) U (A-Z) U (0-9) U _)
  ((a-z) U (A-Z) U (0-9) U _)
  ((a-z) U (A-Z) U (0-9) U _)
  ((a-z) U (A-Z) U (0-9) U _ U ε)
  ((a-z) U (A-Z) U (0-9) U _ U ε)
  ((a-z) U (A·Z) U (0-9) U _ U *ε)*
  ((a-z) U (A-Z) U (0-9) U _ U ε).