Department of Computer Science and Engineering
PES University, Bangalore, India

UE23CS242A: Automata Formal Language and Logic

Prakash C O, Associate Professor, Department of CSE

Regular Language:

- **A regular language is a formal language that can be defined by a regular expression.**
- **A formal language is called regular if it is accepted by a finite state automaton.**

A Regular language can be

1. **described by a regular expression that expresses a regular language as a pattern to which every string in the regular language conforms.**
2. **generated by a regular grammar that generates the strings of the regular language, or**
3. **recognized by a finite-state automaton that accepts the regular language.**

Note:

1. **There is a unique minimal DFA for every regular language**.
2. **Regular expressions and finite automata are equivalent: one can construct a finite automaton that accepts the language defined by a given regular expression, and vice versa.**

Regular Expression (shortened as Regex)

Regular expressions originated in 1951, when mathematician Stephen Cole Kleene described regular languages using his mathematical notation called *regular events*.

Regex Definitions:

- A **regular expression** is **a concise way to describe a set of strings in a language**.
- **A regular expression is an algebraic way to describe a regular language.**
- A **regular expression** is **a sequence of characters that define a search/match pattern in text.** Usually, such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation

Regex Uses:

Regular expressions are used in

1. **search engines,**
2. **search and replace dialogs of word processors and text editors,**
3. **text processing utilities such as sed, grep and AWK,** and
4. **Lexical analysis.**

Note:

- All programming languages provide libraries to handle regular expressions.
- Regular expressions are commonly used in programming languages like Python, JavaScript, Perl, and tools like grep, sed, and AWK.
  1. **Sed** (**stream editor** - most common use of SED command in UNIX is for substitution or for find and replace) is a Unix utility that parses and transforms text.
  2. **AWK** (Aho, Weinberger, and Kernighan) is a domain-specific language designed for text processing and typically used as a data extraction and reporting tool. Like sed and grep, it is a filter, and is a standard feature of most Unix-like operating systems.
     AWK Operations:
     (a) Scans a file line by line
     (b) Splits each input line into fields
     (c) Compares input line/fields to pattern
     (d) Performs action(s) on matched lines
  3. The **grep** (“**global regular expression print**”) **filter searches a file for a particular pattern of characters**, and displays all lines that contain that pattern.)

Regex Construction:

The **regular expressions are built recursively out of smaller regular expressions**. **Each regular expression r denotes a language L(r)**, which is also defined recursively from the languages denoted by r’s subexpressions.

Here are the rules that define the regular expressions over some alphabet ∑ and the languages that those expressions denote.

BASIS: There are two rules that form the basis:

1. ϵ is a regular expression, and L(ϵ) is {ϵ}, that is, the language whose sole member is the empty string.
2. If a is a symbol in ∑, 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.

Regex Common Operators (Core components of Regex)

1) The Concatenation Operator (. or No-character)

- **. (dot) Character (in older representation):**
  This operator concatenates two regular expressions a and b as a.b
  The concatenation symbol . (dot) often is implicit in regular expressions.

- **No Character:**
  This operator concatenates two regular expressions a and b as ab
  No character represents this operator; you simply put *b* after *a*. The result is a regular expression that will match a string if *a* matches its first part and *b* matches the rest. For example, ‘xy’ matches xy.

2) Repetition Operators

Repetition operators repeat the preceding regular expression a specified number of times.

i) The zero-or-more Operator (*)

- This operator **repeats the smallest possible preceding regular expression as many times as necessary** (including zero) **to match the pattern**. '*' represents this operator.
- For example, 'a*' matches any string made up of zero or more a's. Since **this operator operates on the smallest preceding regular expression**, 'fa*' has a repeating 'a', not a repeating 'fa'. So, 'fa*' matches 'f', 'fa', 'faa', and so on.
- The **zero-or-more operator is a suffix operator**, it may be useless as such when no regular expression precedes it. This is the case when it is first in a regular expression, or follows a match-beginning-of-line, open-group, or alternation operator.

ii) The one-or-more Operator (+)

- This operator is similar to the match-zero-or-more operator except that **it repeats the preceding regular expression at least once.**
- For example, supposing that '+' represents the match-one-or-more operator; then 'ca+r' matches, e.g., 'car' and 'caaaar', but not 'cr'.

iii) The zero-or-one Operator (?)

- This operator is similar to the match-zero-or-more operator except that **it repeats the preceding regular expression once or not at all**. In other words, it makes something optional.
- For example, supposing that '?' represents the match-zero-or-one operator; then 'ca?r' matches both 'car' and 'cr', but nothing else.
- Regular expression **a?** represents the regular language: {ε, a}. It accepts either the empty string **ε** or the string '**a**'. This is equivalent to the regular expression: **ε | a**

Note:

1. **The operators *, + and ? are also known as counters or quantifiers.**
2. **The operators * and + are Greedy: Match as Many As Possible (longest match)**
   - By default, quantifiers * and + tells the engine to match as many symbols of its preceding subexpression as possible. This behaviour is called greedy.
   - The matcher processes a match-zero-or-more operator (i.e., *) by first matching as many repetitions of the smallest preceding regular expression as it can. Then it continues to match the rest of the pattern. If it can't match the rest of the pattern, it backtracks (as many times as necessary), each time discarding one of the matches until it can either match the entire pattern or be certain that it cannot get a match.
   - For example, when matching 'ca*ar' against 'caaar', the matcher first matches all three 'a's of the string with the 'a*' of the regular expression. However, it cannot then match the final 'ar' of the regular expression against the final 'r' of the string. So it backtracks, discarding the match of the last 'a' in the string. It can then match the remaining 'ar'.

3) The Alternation (or Union or OR) Operator (| or +(in older representation))

- ➢ The alternation operator (|) **represents a choice between two or more patterns.**
- ➢ If a and b are regular expressions, then **a|b** matches any string that is in the language of either a or b (i.e., **a|b** matches the union of the strings that *a* and *b* match.)

  For example, '|' is the alternation operator, then '**cat|dog|fox**' would match any of 'cat', 'dog' or 'fox'.

- ➢ **The alternation operator operates on the largest possible surrounding regular expressions.** (Put another way, it has the lowest precedence of any regular expression operator.) **Thus, the only way you can delimit its arguments is to use grouping.**

  For example, 'fo(o|b)ar' would match either 'fooar' or 'fobar'. ('foo|bar' would match 'foo' or 'bar'.)

- ➢ **The matcher usually tries all combinations of alternatives so as to match the longest possible string.**

  For example, when matching '**(fooq|foo)*(qbarquux|bar)**' against 'fooqbarquux', it cannot take, say, the first combination it could match, since then it would be okay to match just 'fooqbar'.

4) Grouping Operators ((…))

- ➢ **A group, also known as a subexpression, consists of an open-group operator (left parenthesis), any number of other operators, and a close-group operator (right parenthesis).**
- ➢ **Regex treats this sequence as a unit**, just as mathematics and programming languages treat a parenthesized expression as a unit.
- ➢ **Therefore, using *groups*, you can:**
  - o **delimit the argument(s) to an alternation operator** or **a repetition operator.**
  - o keep track of the indices of the substring that matched a given group.

Regex operators precedence

The following table gives the order of RE operator precedence, from highest precedence to lowest precedence.

Highest	**[]**	**Character class**
	()	**Grouping**
	*** + ? {m, n}**	**Repetition Operators**
	. or **No-character operator**	**Concatenation**
	^ $	**Position Anchors**
Lowest	**|**	**Alternation (or Union)**

Note:

- ➢ **Anchors (^ and $)** are special characters **that anchor regular expressions to particular places in a string.** The most common anchors are the caret ^ and the dollar sign $.
- ➢ \ (backlash) is used to suppress the special meaning of a character when matching.
  For example: \$ matches the character '$'.
- ➢ **Regex Characters**: These characters have special meaning in regex . + * ? ^ $ ( ) [ ] { } | \

Regex in Practice

Character classes ([…])

- A regular expression **$a_1|a_2|$ • • • $|a_n$**, where the a_i's are each symbol of the alphabet, can be replaced by the shorthand **[a_1a_2 … a_n]**
- When **a_1**, **a_2**, **…** ,**a_n** form a logical sequence, we can replace them by **a_1-a_n**. Thus, **[abc] is shorthand for a|b|c**, and **[a-z] is shorthand for a|b|….|z**

Expression	Description
[…]	• Denotes the character class. • Matches one of the characters in the brackets. • Example: T[ao]p Sample match: Tap or Top
[x-z]	• Matches one of the characters in the range from x to z • Matches x, or y, or z. Shorthand for x|y|z
[xyz]	Matches x, or y, or z
[x\-z]	Matches x, or -, or z
[A-Za-z0-9]+	Matches one or more alphanumeric characters
[^x]	Matches any character except x
[^x-z]	• Matches one of the characters not in the range from x to z • (Anything except x or y or z) • The ^ prefix in a character class creates a *complement class*.
[x^z]	One of x, ^, z
[x|z]	One of x, |, z
.	Matches any (single) character except newline
^x	• (Caret.) Matches x at the start of the string or after newline. • ^ indicates start of string or after newline. • Regex ^Hello matches any string that starts with "Hello", like "Hello World" but not "Say Hello".
z$	• Matches z at the end of a string or just before the newline. • $ indicates end of string or before newline. • Ensures the pattern appears only at the end. • Anchors in regular expressions are powerful tools used to match positions within a string.
^$	Matches empty string

Note: The caret ^ has three uses:

1. to match at the start of a line or string,
2. to indicate a negation, if it is the first character inside square brackets(i.e., character class), and
3. just to mean a caret.

Expression	Description
x{n}	Matches exactly n occurrences of x
x{m,n}	Matches m through n occurrences of x.
x{n,}	Matches n or more occurrences of x Examples: a{0, } -> same as a*, a{1, } -> same as a+
xx|yy	Matches either xx or yy
(x)	Matches x
x/y	Matches x but only if followed by y

The programming languages provide macros (or meta-characters)

Expression	Description
.	Any one character except newline. Same as [^\n]
\d	Matches any digit between [0-9] Examples: \d{4} matches any 4 digits \d{3,4} matches 3 or 4 digits
\w	Matches any ASCII letter, digit or underscore
\s	Matches any space, tab, newline, carriage return, vertical tab

Operators	Description
?	Zero or one copy of the preceding expression
*****	Zero or more copies of the preceding expression
+	One or more copies of the preceding expression

Examples:

Expression	Description
a|b	a or b (alternating)
(ab)+	One or more copies of ab (grouping)
abc	abc
abc*	ab abc abcc abccc abcccc …
abc+	abc abcc abccc abcccc …
a(bc)+	abc abcbc abcbcbc …
a(bc)*	a abc abcbc abcbcbc …

a(bc)?	a abc (none or once)

Example 1:

Let Σ = {a, b, c}. Regular expressions for

a) single letter words

b) words that contain at least one a.

c) words that contain at least one a or one b.

d) words that contain at least one a and one b.

Solution:

a) **a|b|c** **or** [abc]

b) (a|b|c)* a (a|b|c)* **or** [abc]* a [abc]*

c) (a|b|c)* (a|b) (a|b|c)* **or** (a|b|c)* [ab] (a|b|c)*

d) (a|b|c)* a (a|b|c)* b (a|b|c)* | (a|b|c)* b (a|b|c)* a (a|b|c)*

Example 2:

Write regex to find/validate C-language

a) Identifiers

b) Integers (with ±0), floating-point numbers and E-notation numbers

Solution:

a) `^[a-zA-Z_] ([a-zA-Z0-9_])*$`

b) **Integers:** `[+-]? 0 | [+-]?[1-9][0-9]*` or

`[+-]? (0 | [1-9][0-9]*)` or

`[+-]? 0 | [+-]?[1-9]\d*`

We did not use *position anchors* ^ and $ in this regex. Hence, it can match any parts of the input string. For examples,

- If the input string is "abc123xyz", it matches the substring "123".
- If the input string is "abcxyz", it matches nothing.
- If the input string is "abc123xyz456_0", it matches substrings "123", "456" and "0" (three matches).
- If the input string is "0012300", it matches substrings: "0", "0" and "12300" (three matches)!!!

`^[+-]? 0 | [+-]?[1-9][0-9]*$` or

`^[+-]? (0 | [1-9][0-9]*)$` or

`^[+-]? 0 | [+-]?[1-9]\d*$`

This regex matches an Integer literal (for entire string with the *position anchors*), both positive, negative and zero.

Floating point numbers: `^[+-]? (0|[1-9][0-9]*) \. [0-9]+$`

E-notation numbers: `^[0-9] \. [0-9]+ E[+-]?[0-9]+$`

Note: Signed zero is zero with an associated sign. In ordinary arithmetic, the number 0 does not have a sign, so that −0, +0 and 0 are equivalent. However, in computing, some number representations allow for the existence of two zeros, often denoted by −0 (negative zero) and +0 (positive zero).

Exercise:

1. Let Σ = {0, 1}. Write regular expressions to find

a) strings that contain only 0's and string length longer than 2

b) strings that do not contain 11.

Solution:

a) `0{3,}`

b) Many solutions

- `(0|10)* | (0|01)* | 1(0|01)*`
- `(ε|1)(0|01)*`
- `(1|ε)(0(1|ε))*`
- `((1|ε)0)*(1|ε)`
- `(1|ε)(00*1)*0*`

2. Write regex to validate

a) **PAN Number**
(A ten-character alphanumeric identifier, the first five characters are letters (in uppercase by default), followed by four numerals, and the last (tenth) character is a uppercase letter)

b) **Adhaar Number with space after 4-digits**
(It should have 12 digits. It should not start with 0 and 1. It should not contain any alphabet and special characters.)

c) **Mobile Number with Optional Country Code**
(All mobile phone numbers are 10 digits long, In India, mobile numbers start with either 9, 8, 7 or 6.)

d) **Validate**

i) **Date format dd/mm/yyyy** or **dd-mm-yyyy**, and days between 1-31, months between 1-12 and 4-digits year)

ii) **Year 2025 dates**

e) **Email Address**

f) **IPv4 Address**

Solution:

a) `^[A-Z]{5}[0-9]{4}[A-Z]$`

b) `^[2-9][0-9]{3}\s[0-9]{4}\s[0-9]{4}$` or `^[2-9]\d{3}\s\d{4}\s\d{4}$`

c) `^(0|\+91)?[6-9]\d{9}$`

d) (i) `^(0?[1-9]|[12][0-9]|3[01])[\/\-](0?[1-9]|1[012])[\/\-]\d{4}$`

(ii) ...

e) `^\w+([.\-]?\w+)*@\w+([.\-]?\w+)*(\.\w{2,3})+$`

Explanation:

- **The *position anchors* ^ and $ match the beginning and the ending of the input string, respectively. That is, this regex shall match the entire input string, instead of a part of the input string (substring).**
- \w+ matches 1 or more word characters (same as [a-zA-Z0-9_]+).
- [.-]? matches an optional character . or -. Although dot (.) has special meaning in regex.
  In addition to ^,] and \ , the following characters must be escaped in character classes if they represent literal characters: (,) , [, { , } , / , - , |
- The sub-expression \w+([.-]?\w+)* is used to match the username in the email, before the @ sign. It begins with at least one word character [a-zA-Z0-9_], followed by more word characters or . or -. However, a . or - must follow by a word character [a-zA-Z0-9_]. That is, the input string cannot begin with . or -; and cannot contain "..", "--", ".-" or "-.". Example of valid string are "a.1-2-3".
- The @ matches itself. In regex, all characters other than those having special meanings matches itself, e.g., a matches a, b matches b, and etc.
- Again, the sub-expression \w+([.-]?\w+)* is used to match the email domain name, with the same pattern as the username described above.
- The sub-expression \.\w{2,3} matches a . followed by two or three word characters, e.g., ".com", ".edu", ".us", ".uk", ".co".
- (\.\w{2,3})+ specifies that the above sub-expression could occur one or more times, e.g., ".com", ".co.uk", ".edu.sg" etc.

f) **^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$**

3. Let Σ = {0, 1}. Give regular expressions for

a) words such that no prefix have the difference between the number of 0s and 1s more than 1.
b) words such that no prefix have the difference between the number of 0s and 1s more than 2

Regular Expression to Finite Automata

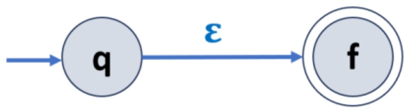
Converting regular expressions to finite automata using Thompson Construction Method:

Thompson Construction Method:

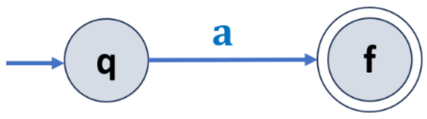
The algorithm works recursively by splitting an expression into its constituent subexpressions, from which the NFA will be constructed using a set of rules.

Following are the rules:

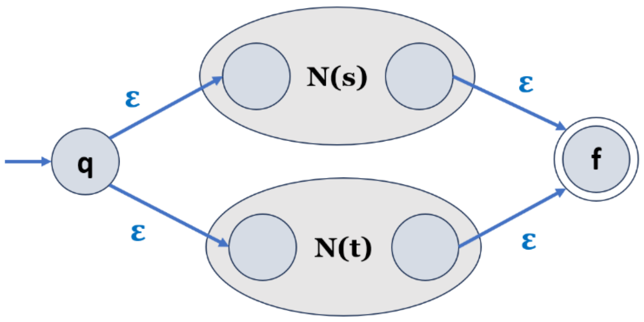
- **If subexpression operand is epsilon(ε), then our FA has two states, q (the start state) and f (the final, accepting state), and an ε-transition from q to f.**



- **If subexpression operand is a character a, then our FA has two states, q (the start state) and f (the final, accepting state), and a transition from q to f with label a.**

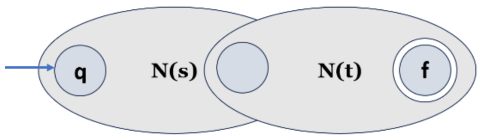


- **If subexpression is the alternation (or union) expression s|t, then our FA is**



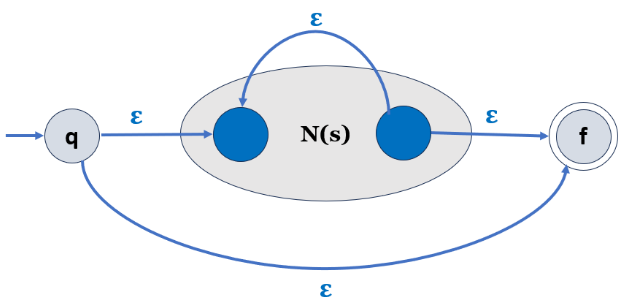
State q goes via ε either to the initial state of Automata of s and t. N(s) or N(t). Their final states become intermediate states of the whole NFA and merge via two ε-transitions into the final state of the NFA.

- **If subexpression is the concatenation expression st, then our FA is**



The initial state of N(s) is the initial state of the whole NFA. The final state of N(s) becomes the initial state of N(t). The final state of N(t) is the final state of the whole NFA.

- **If subexpression is the Kleene star expression s*, then our FA is**



An ε-transition connects the initial and final state of the NFA with the sub-NFA. N(s) Automata for s in between. Another ε-transition from the inner final to the inner initial state of N(s) allows for repetition of expression s according to the star operator.

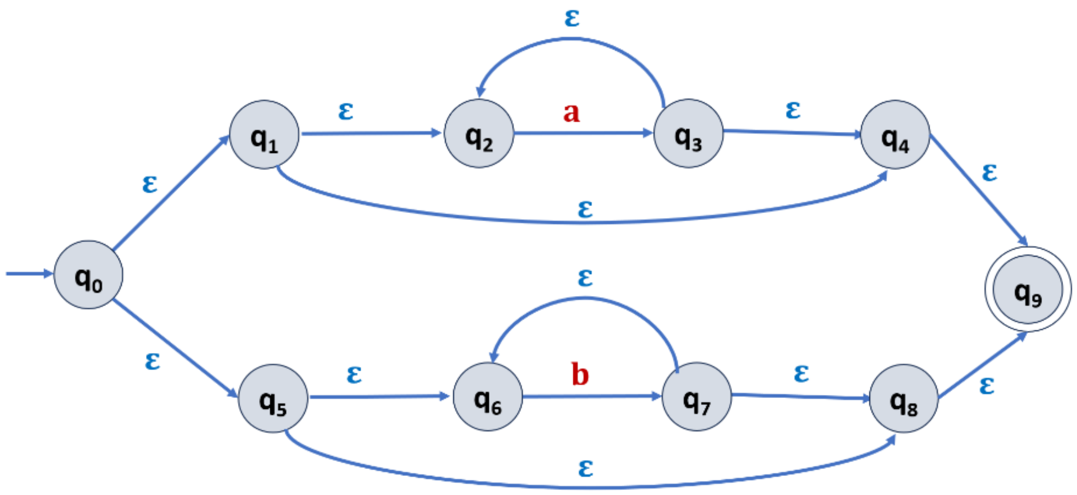
- **The parenthesized expression (s) is converted to N(s) itself.**

With these rules, using the empty expression and symbol rules as base cases, it is possible to prove with mathematical induction that any regular expression may be converted into an equivalent NFA

Exercise: Construct Finite Automata for the following Regular Expressions using Thompson construction method.

```
1) (ab|c)*
2) (ab|ba)*
3) a*|b*
4) (0|01*0)*0
```

`Solution for a*|b*`



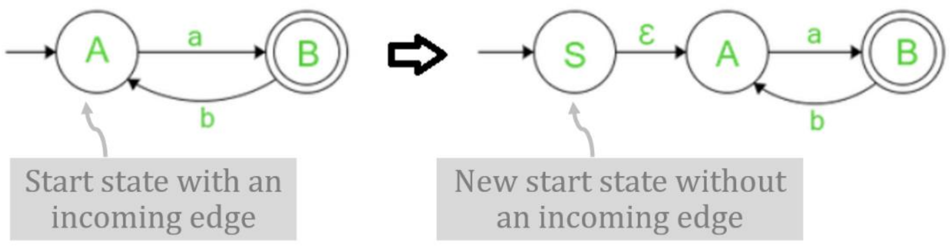
FA to Regex: State Elimination Method

This method involves the following steps in finding the regular expression for any given Finite Automata.

Step-01: (Ensure a start state with no incoming transitions)

- If there are no incoming edges to the start state proceed further to check other rules.
- If there are any incoming edges (or transitions) to the start state, to get rid of the incoming edges, make new start state with no incoming edges and then make an outgoing edge from new start state to the old start state with Ɛ-transition.
  The initial state before is now normal state with added incoming Ɛ-transition.

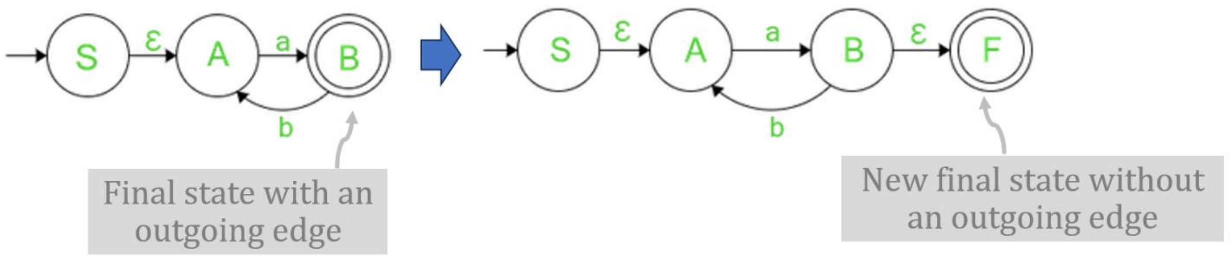
Example:


A
a
B
b
S
Ɛ
A
a
B
b
Start state with an incoming edge
New start state without an incoming edge

Step-2a: (Ensure a single final state with no outgoing transitions)

- If there exists one final state and if there are any outgoing edges from the final state, then create a new final state (having no outgoing edge from it) and then make an outgoing edge from the previous final state to the new final state with Ɛ-transition.
- The previous final state is now normal non-final state with an added outgoing Ɛ-transition.

Example:

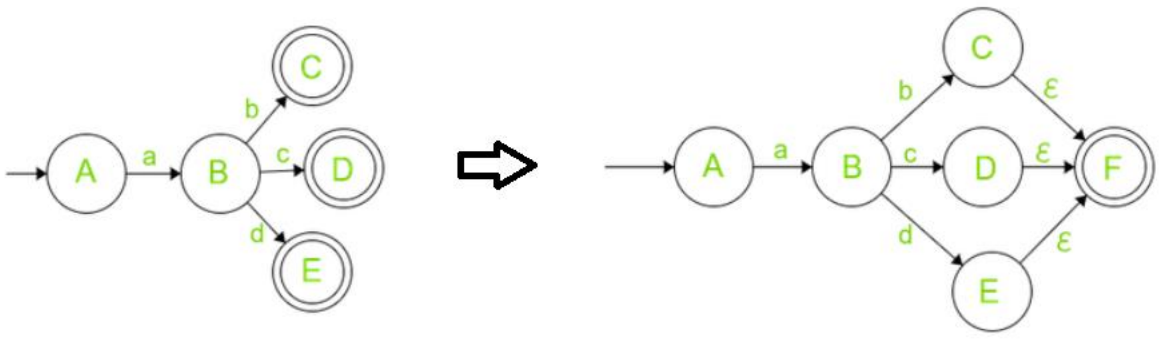


S
ε
A
a
B
b
Final state with an outgoing edge
S
ε
A
a
B
ε
F
b
New final state without an outgoing edge

Step-2b: (Ensure a single final state with no outgoing transitions)

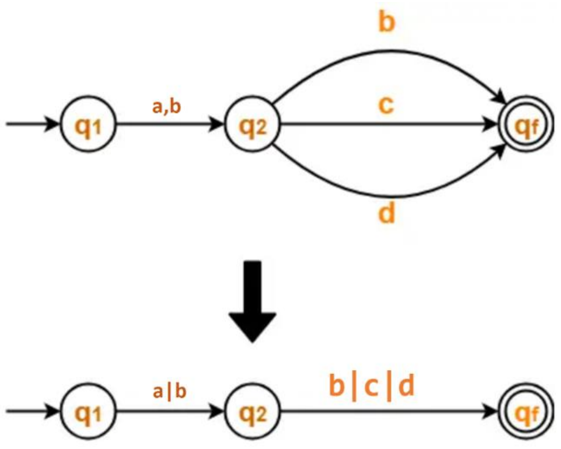
- If there exist multiple final states in the DFA, then convert all the final states into non-final states and create a new single final state and then make an outgoing edge from all the previous final states to the new final state with Ɛ-transition.
- The previous final states are now normal non-final states with added outgoing Ɛ-transition.

Example:



Step-03: Represent Transitions as Regex

- Replace each transition between states with a regular expression.
- If multiple transitions exist between two states, combine them using | (union).



Step-04:

- Eliminate all the intermediate states one by one (Until Only Start and Final States Remain).
- These states may be eliminated in any order or by following some order.

For each state to be eliminated: (say state q_2)

- Identify all **incoming** and **outgoing** transitions of state q_2.
- For each pair of states (q_1 q_3) that pass through the state q_2 being eliminated:
  If $q_1 \to q_2$ is labeled with r_1, $q_2 \to q_2$ is labeled with r_2 (self loop), and $q_2 \to q_3$ is labeled with r_3, then: **New transition from** $q_1 \to q_3$ (after eliminating state q_2)is labeled with $r_1\ r_2^*\ r_3$

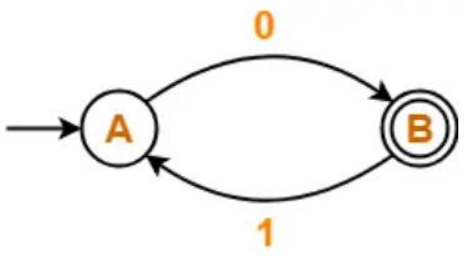
In the end,

- Only an initial state going to the final state will be left.
- The cost of this transition is the required regular expression.

Examples:

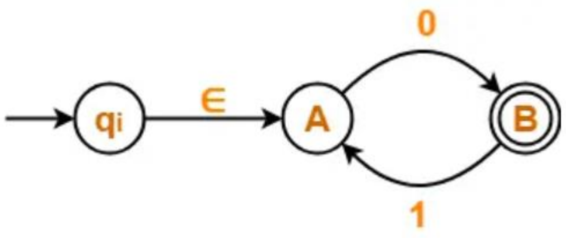
1) Find regular expression for the following FA using state elimination method.

Note: The State elimination order is A and then B.

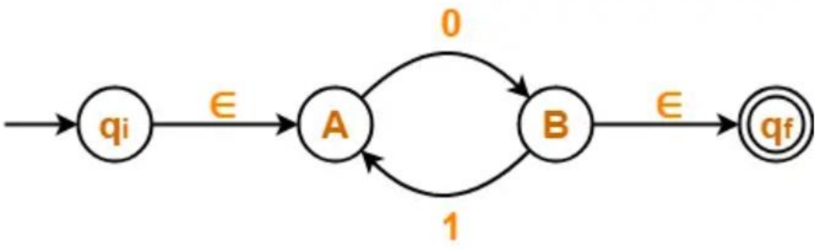


Solution:

Step-01: Initial state A has an incoming edge. So, we create a new initial state q_i. The resulting FA is



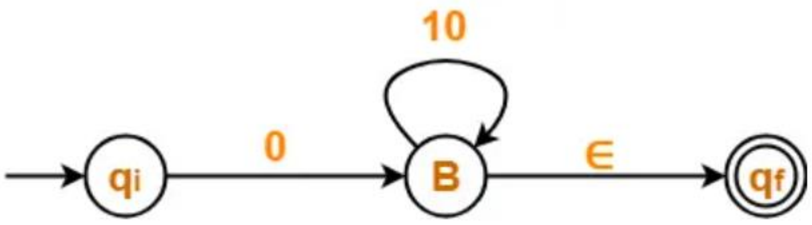
Step-02: Final state B has an outgoing edge. So, we create a new final state q_f. The resulting FA is



Step-03: Now, we start eliminating the intermediate states. First, let us eliminate state A.

- There is a path going from state q_i to state B via state A.
- So, after eliminating state A, we put a direct path from state q_i to state B having cost **ε.0 = 0**
- There is a loop on state B using state A.
- So, after eliminating state A, we put a direct self-loop on state B having cost **1.0 = 10**

Eliminating state A, we get

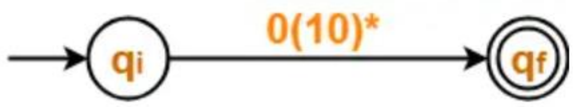


Step-04:

Now, let us eliminate state B.

- There is a path going from state q_i to state q_f via state B.
- So, after eliminating state B, we put a direct path from state q_i to state q_f having cost 0.(10)*.ε = 0(10)*

Eliminating state B, we get



From here,

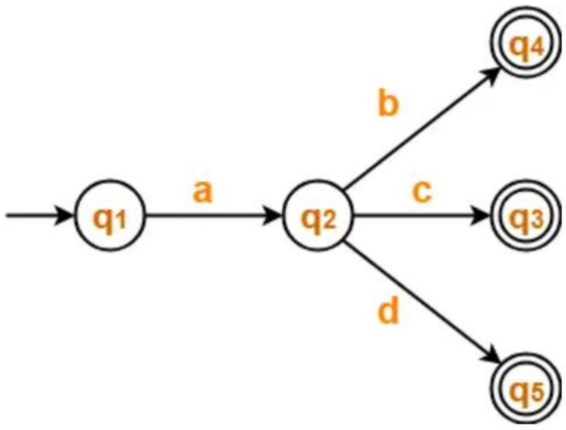
Regular Expression = 0(10)*

Note:

In the above question, If we first eliminate state B and then state A, then regular expression would be = (01)*0. This regex is also representing the same FA and it also correct.

2) Find regular expression for the following FA using state elimination method.

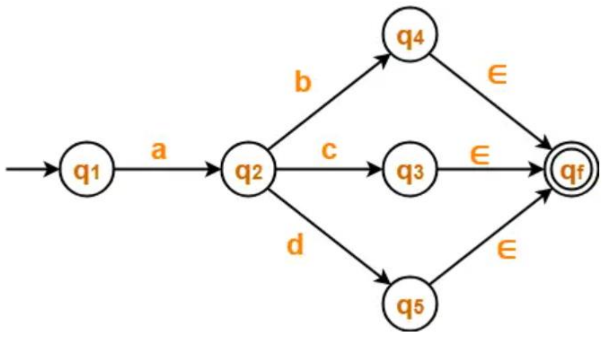
Note: The State elimination order is q4, q3, q5 and then q2.



Solution:

Step-01: There exist multiple final states. So, we convert them into a single final state.

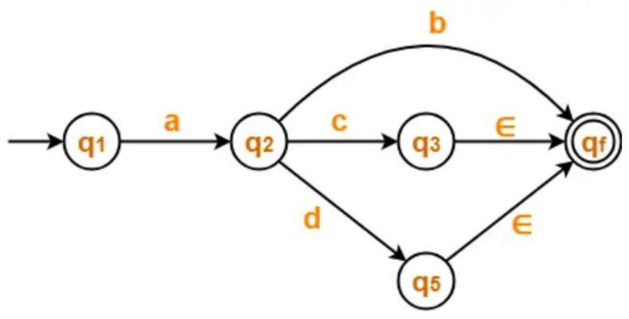
The resulting FA is



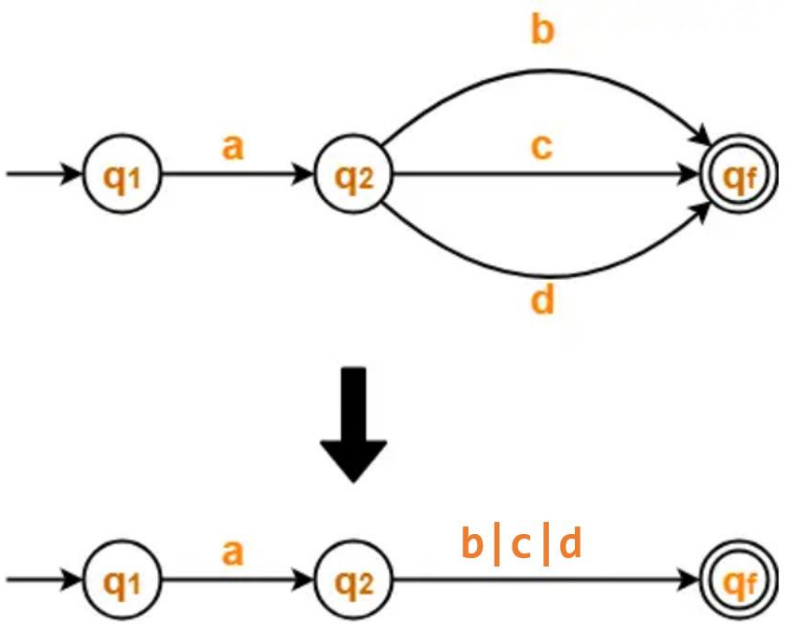
Step-02: Now, we start eliminating the intermediate states.

First, let us eliminate state q_4.

- There is a path going from state q_2 to state q_f via state q_4.
- So, after eliminating state q_4 , we put a direct path from state q_2 to state q_f having cost b.ε = b.

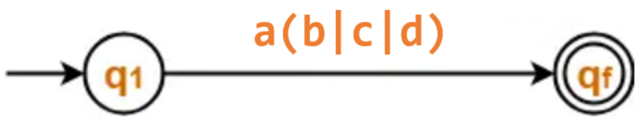


Similarly, eliminate q3 and q5 a,b



Step-05: Now, let us eliminate state q_2.

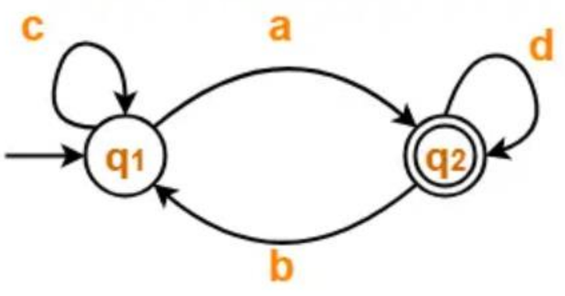
- There is a path going from state q_1 to state q_f via state q_2.
- So, after eliminating state q_2 , we put a direct path from state q_1 to state q_f having cost a.(b|c|d).



Regular Expression = a(b|c|d)

3) Find regular expression for the following FA using state elimination method.

Note: The State elimination order is q1 and then q2.

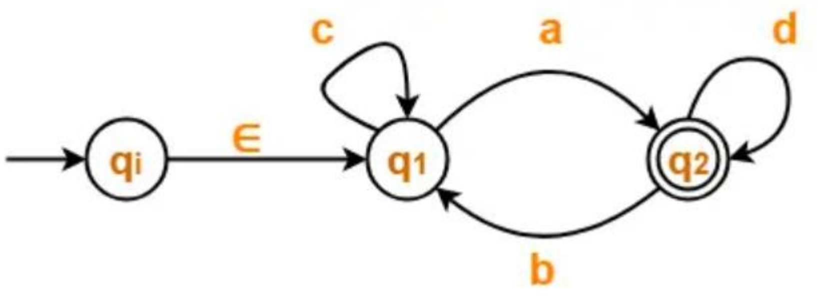


Solution:

Step-01:

- Initial state q_1 has an incoming edge.
- So, we create a new initial state q_i.

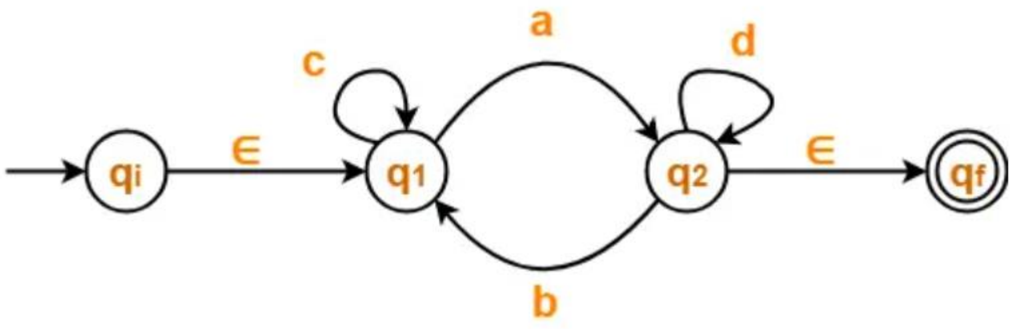
The resulting FA is



Step-02:

- Final state q_2 has an outgoing edge.
- So, we create a new final state q_f.

The resulting FA is



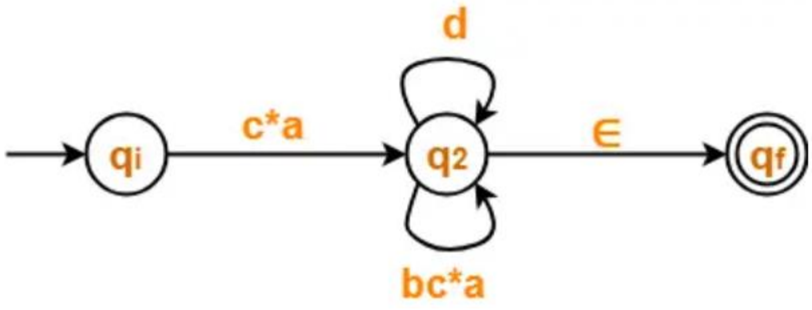
Step-03:

Now, we start eliminating the intermediate states.

First, let us eliminate state q_1.

- There is a path going from state q_i to state q_2 via state q_1 .
- So, after eliminating state q_1, we put a direct path from state q_i to state q_2 having cost $\varepsilon.c^*.a = c^*a$
- There is a loop on state q_2 using state q_1 .
- So, after eliminating state q_1 , we put a direct loop on state q_2 having cost $b.c^*.a = bc^*a$

Eliminating state q_1, we get

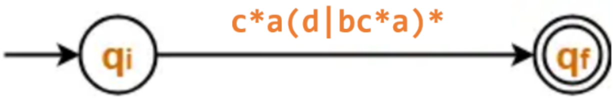


Step-04:

Now, let us eliminate state q_2.

- There is a path going from state q_i to state q_f via state q_2 .
- So, after eliminating state q_2, we put a direct path from state q_i to state q_f having cost c*a(d|bc*a)*ε = c*a(d|bc*a)*

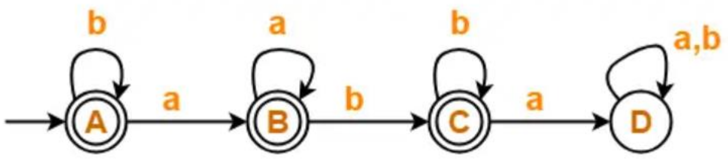
Eliminating state q_2, we get



Regular Expression = c*a(d|bc*a)*

4) Find regular expression for the following DFA using state elimination method.

Note: The State elimination order is C, B and then A.



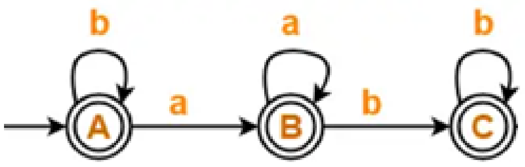
- State D is a dead state as it does not reach to any final state.
- So, we eliminate state D and its associated edges.

Solution:

Step-01:

- State D is a dead state as it does not reach to any final state.
- So, we eliminate state D and its associated edges.

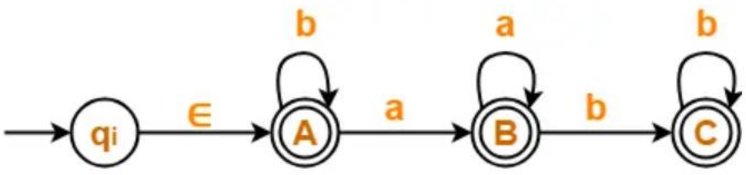
The resulting DFA is



Step-02:

- Initial state A has an incoming edge (self loop).
- So, we create a new initial state q_i.

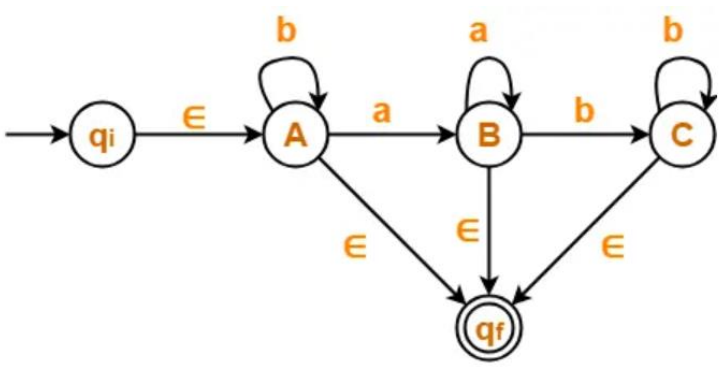
The resulting FA is



Step-03:

- There exist multiple final states.
- So, we convert them into a single final state.

The resulting FA is



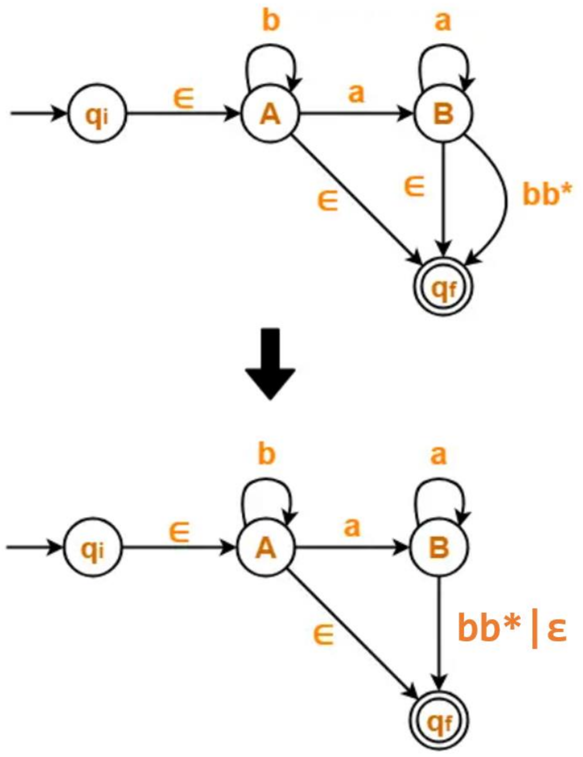
Step-04:

Now, we start eliminating the intermediate states.

First, let us eliminate state C.

- There is a path going from state B to state q_f via state C.
- So, after eliminating state C, we put a direct path from state B to state q_f having cost b.b*.ε = bb*

Eliminating state C, we get

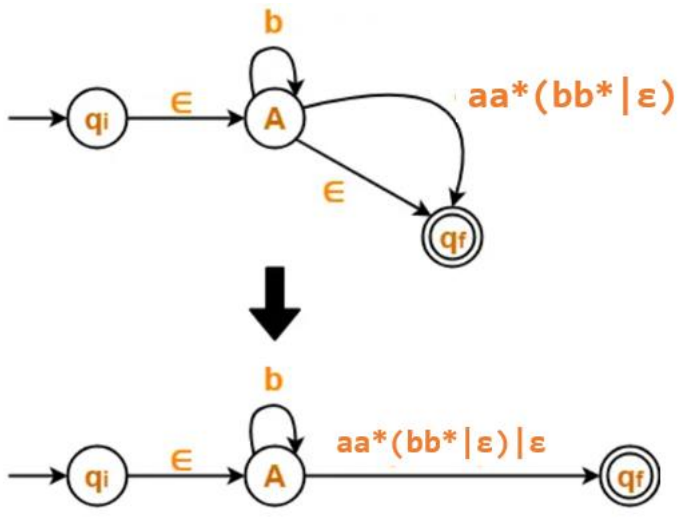


Step-05:

Now, let us eliminate state B.

- There is a path going from state A to state q_f via state B.
- So, after eliminating state B, we put a direct path from state A to state q_f having cost a.a*.(bb*|ε) = aa*(bb*|ε)

Eliminating state B, we get

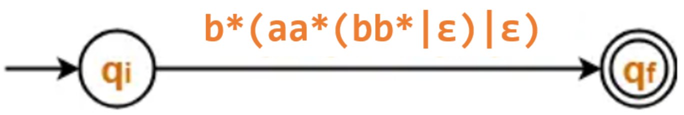


Step-06:

Now, let us eliminate state A.

- There is a path going from state q_i to state q_f via state A.
- So, after eliminating state A, we put a direct path from state q_i to state q_f having cost ε.b*.(aa*(bb*|ε)|ε) = b*(aa*(bb*|ε)|ε)

Eliminating state A, we get



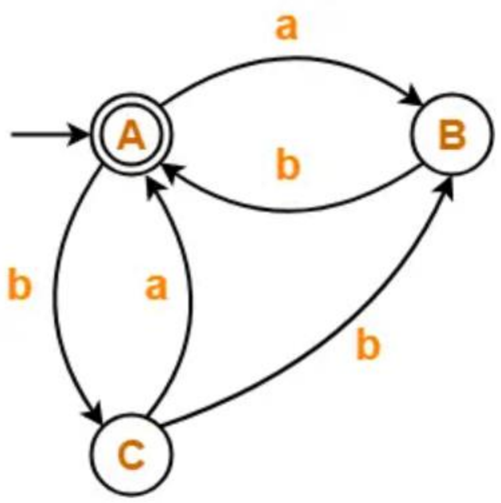
Regular Expression = b*(aa*(bb*|ε)|ε)

We know, bb* | ε = b*   So, we can also write

Regular Expression = b*(aa*b*|ε)

5) Find regular expression for the following DFA using state elimination method.

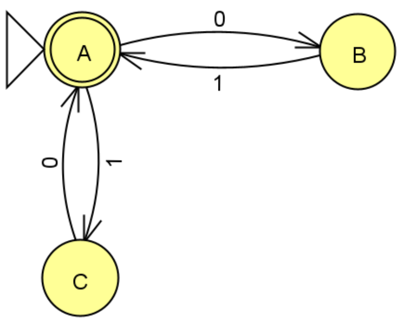
Note: The State elimination order is B, C and then A.



Regular Expression = (ab | b(a|bb))*

6) Find regular expression for the following DFA using state elimination method.

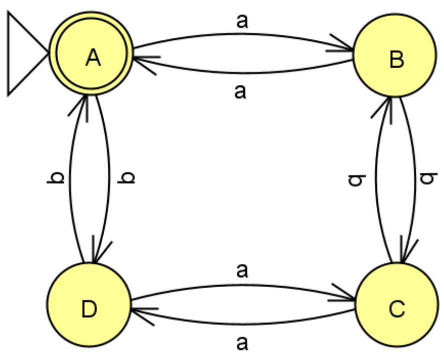
Note: The State elimination order is B, C and then A.



Regular Expression = (01|10)*

7) Find regular expression for the following DFA using state elimination method.

Note: The State elimination order is B, D, C and then A.



Regular Expression = (((ab|ba) (aa|bb)* (ab|ba)) | (aa|bb))*

Rules for regular expressions:

The set of regular expressions is defined by the following rules.

- **Every letter of ∑ can be made into a regular expression, null string, ε itself is a regular expression.**
- **If r1 and r2 are regular expressions, then (r1), r1r2, r1|r2, r1* , r^+ are also regular expressions.**

Example: ∑ = {a, b} and r is a regular expression of language made using these symbols

Regular expression	Regular language
∅ (empty set)	L(∅) = ∅
ε (empty string)	L(ε) = {ε}
a* (Kleene closure)	L(a*) = {ε, a, aa, aaa .....}
a|b (Union)	L(a|b) = L(a) U L(b) = {a, b}
ab (concatenation)	L(ab) = L(a) L(b) = {ab}
a* | ba	L(a* | ba)= L(a*) U L(ba) = {ε, a, aa, aaa,...... , ba}

Algebraic properties of regular expressions:

- Kleene star and Kleene plus are unary operators.
- Union or alternation(|) and concatenation operators (. or no-character) are binary operators.

1. Closure Property:

If r_1 and r_2 are regular expressions (REs), then

- r_1^* is a Regular expression (i.e., Kleene star (*) satisfies Closure property)
- $r_1|r_2$ or $r_1 \cup r_2$ is a Regular expression (i.e., Alternation (|) satisfies Closure property)
- r_1r_2 is a Regular expression (i.e., Concatenation satisfies Closure property)

Note: In mathematics, the closure property states that when **any operation is performed on elements within a set, the result is also within the same set.**
Closure property states that when a set of numbers is closed under any arithmetic operation such as addition, subtraction, multiplication, and division, it means that when the operation is performed on any two numbers of the set with the answer being another number from the set itself.

Closure laws

- $(r^*)^* = r^*$ (closing an expression that is already closed does not change the language)
- $\varepsilon^* = \varepsilon$ (a string formed by concatenating any number of copies of an empty string is empty itself)
- $r^+ = rr^* = r^*r$, as $r^* = \varepsilon \mid r \mid rr \mid rrr \ldots$ and $rr^* = r \mid rr \mid rrr \ldots$
- $r^* = r^+ \mid \varepsilon$
- $$\begin{aligned}(r_1|r_2)^* &= (r_1^*|\, r_2^*)^* \\ &= (r_1^*\, r_2^*)^* \\ &= (r_1|\, r_2^*)^* \\ &= (r_1^*|\, r_2)^* \\ &= r_1^*(\, r_2\, r_1^*)^* \\ &= r_1^*(\, r_1\, r_2^*)^*\end{aligned}$$

3. Associativity

The **associativity property** states that the **grouping of operations does not affect the result, as long as the operations are associative.**

If r_1, r_2, r_3 are RE, then

i) $r_1|(r_2|r_3) = (r_1|r_2)|r_3$

For example: $r_1 = a$, $r_2 = b$, $r_3 = c$, then

- The resultant regular expression in LHS becomes **a|(b|c)** and the regular set for the corresponding RE is {a, b, c}.
- for the RE in RHS becomes **(a|b)|c** and the regular set for this RE is {a, b, c}, which is same in both cases. Therefore, **the associativity property holds for union operator**.

ii) $r_1(r_2r_3) = (r_1r_2)r_3$

- **For example:** $r_1 = a$, $r_2 = b$, $r_3 = c$
- Then the string accepted by RE **a(bc)** is only abc.

- The string accepted by RE in RHS is `(ab)c` is only abc ,which is same in both cases.

  Therefore, the **associativity property holds for concatenation operator.**

iii) Associativity property does not holds for Kleene star(*) and Kleene plus(+) because they are unary operators.

4. Identity Property

The identity property **states that there exists an element (called the identity element) in a set such that when you apply an operation to any element and the identity, the result is the original element.**

- **In the case of alternation (or union) operator**

  `r|∅ = r  as r∪∅ = r`

  Therefore, **∅ is the identity element for a union operator.**

- **In the case of concatenation operator**

  $r\varepsilon = r$

  Therefore, **ε is the identity element for concatenation operator.**

6. Commutative Property

An operation is commutative if changing the order of the operands does not change the outcome.

If r_1, r_2 are RE, then

- $r_1|r_2 = r_2|r_1$ For example, for r_1 =a and r_2 =b, then RE's a|b and b|a are equal.
- $r_1r_2 \neq r_2r_1$ For example, for r_1 = a and r_2 = b, then **regex ab is not equal to ba.**

7. Distributive Property

We can **simulate** distributive behavior using **grouping**, **concatenation** and **alternation operators**.

If r_1, r_2, r_3 are regular expressions, then

- $r_1(r_2|r_3) \equiv r_1r_2|r_1r_3$

  Distributive property says that **concatenation distributes over union**.

- $(r_1|r_2)r_3 \equiv r_1r_3|r_2r_3$

  The distributive property also applies when the union is on the left.

- $r_1|(r_2r_3) \neq (r_1|r_2)(r_1|r_3)$

  Distributive property says that **union does not distribute over concatenation**.

The **Distributive Property** is a fundamental rule in mathematics that helps to simplify expressions and solve equations.

8. Idempotent Property

- $r_1|r_1 = r_1 \Rightarrow r_1 \cup r_1 = r_1$, therefore the **union operator satisfies idempotent property**.

  `(cat|cat) ≡ cat`

- $r_1r_1 \neq r_1$ ⇒ **concatenation operator does not satisfy idempotent property**.

Note: In mathematics, a number that is **idempotent keeps the same value when operated by itself**.

In regex, a pattern is idempotent if repeating it doesn't change what it matches. That is, applying the same pattern multiple times has the same effect as applying it once

1) `a* ≡ a*a* ≡ a*a*a*...`
2) `(abc)?(abc)? ≡ (abc)?`

Why It Matters: Helps to simplify regex patterns. Avoids redundant computation and makes patterns more readable and maintainable

9. Identities for regular expression

There are many identities for the regular expression. Let p, q and r be regular expressions.

- `∅|r = r|∅ = r`

  `∅∪r = r∪∅ = r`

  That is, Union of the null set to any other language will not change it.

- `∅r = r∅ = ∅`

  That is, we cannot concatenate non empty language L(r) with a language ∅ containing no string (as there is no string to concatenate in ∅), it yields empty language.

- `εr = rε = r`

  That is, concatenating the blank (i.e., empty) string to any string will not change it.

- `∅* = ε`

  Note:

  1) **What happens when you apply * to ∅?**

     Let's think through it:

     - ∅* means: "zero or more strings from the empty set."
     - But the empty set has **no strings** to repeat.
     - So the **only valid repetition** is **zero times**, which gives you... **ε**!

     Final Result: ∅* = {ε}

     So in regex terms, ∅* matches **only the empty string**, which is exactly what **ε** represents.

     It's a beautiful little corner of formal language theory — **even nothing, repeated zero times, still gives you something: the empty string**. Want to go deeper into regex identities or automata theory?

  2) For any language L, by definition

     $$L^* = \bigcup_{i=0}^{\infty} L^i,$$

     where a word in L^i is the concatenation of i words from L. In particular, **$L^0=\{\epsilon\}$** since ϵ is the **concatenation of zero words from L. It doesn't matter if L is empty or not**, since we are choosing *zero* words from L.

- `r|r = r`
- `rr* = r*r = r⁺`
- `ε|rr* = ε|r⁺ =r*`

- (pq)*p = p(qp)*
- (p|q)* = (p*q*)* = (p*|q*)*
- (p|q)r= pr|qr and r(p|q) = rp | rq

Exercise-1: Find regular expressions for the following languages:

1. The set of all strings with an even number of 0's
2. The set of all strings of even length (length multiple of k)
3. The set of all strings that begin with 110
4. The set of all strings containing exactly three 1's
5. The set of all strings divisible by 2
6. The set of strings where third last symbol is 1

Exercise-2: Write regular expression to denote a language L

a) String which begin or end with either 00 or 11.

b) The set of all strings, when viewed as binary representation of integers, that are divisible by 2.

c) The set of all strings containing 00.

d) String not containing the substring 110.

Answer:

```
a) (00|11) (0|1)* | (0|1)* (00|11)
b) (0|1)*0      i.e all strings ending with 0
c) (0|1)*00 (0|1)*
d) (0|10)*1*
```

Regex and its Language

Look for each regular expression e which language L(e) does it produce.

(1) What about the regex *a* (only the letter)? Its language is

L(a) = {a}, just the single word/character "a".

(2) For the regex e_1 | e_2, where e_1 and e_2 are regexs themselves,

$L(e_1|e_2) = L(e_1) \cup L(e_2)$.

Example: L(a|b) = {a, b}.

(3) **For the regex e_1e_2 (concatenation), where e_1 and e_2 are regexs themselves,**

$L(e_1e_2)$ = all words w such that we can write $w = w_1w_2$ with w_1 in $L(e_1)$ and w_2 in $L(e_2)$.

(4) **What about a regular expression e*, where e might be a regular expression itself?**

Intuitively, a word is in L(e*) if it has the form $w_1w_2w_3w_4 \dots w_n$, with w_i in L(e) for each i. So

L(e*) = all words w such that we can write

$w = w_1w_2 \dots w_n$

for n >= 0 with all w_i in L(e) (for i = 1, 2, ..., n)

So, what about `L(a*|b*)?`

```
L(a*|b*) = L(a*) U L(b*)
         = { ε, a, aa, aaa, aaaa, ....} U { ε, b, bb, bbb, bbbb}
```
= all strings that have either only a's OR only b's in it (including ε)

Similarly for `(a*b*):`

```
 L(a*b*) = all words w = w₁w₂ with w₁ in L(a*) and w₂ in L(b*)
         = { ε, a, aa, aaa, ..., b, bb, bbb, …, ab, aab, abb, aabb, …}
```
= all strings that have zero or more a's, zero or more b's, a's followed by b's

Equivalence of two regular expressions

We say that two regular expressions R and S are equivalent if they describe the same language. In other words, if L(R) = L(S) for two regular expressions R and S then R = S.

We can determine the equivalence of two regular expressions R and S using:

1) **A Formal method**
2) **An Informal method**

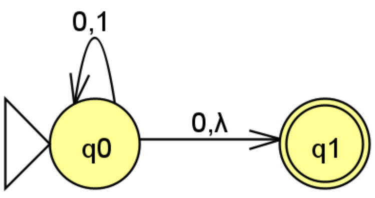
1) Formal method to prove equivalence of two regular expressions

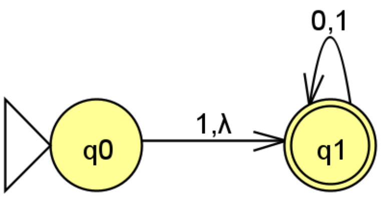
1) Let R and S be two regular expressions.
2) Construct NFA M(R) for regular expression **R** and NFA M(S) for regular expression **S.**
3) Convert both the NFAs M(R) and M(s) to respective DFAs using subset construction method.
4) Next, find the Minimal DFAs for the two DFAs constructed. Check if the resulting automata are isomorphic (structurally identical).
5) If two Minimal DFAs are equivalent, then regular expressions **R** and **S** are equivalent else **R** and **S** are not equivalent.

Example:

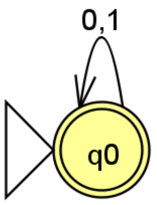
Consider the two regular expressions **R** = **(0|1)* (0|λ)** and **S** = **(1|λ) (0|1)***

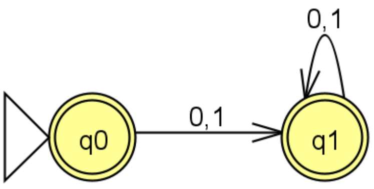
NFAs:



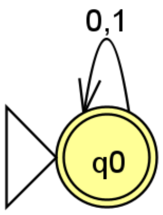


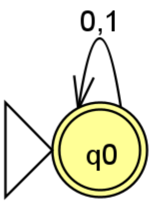
DFAs:





Minimal DFAs:





Check if the resulting automata are isomorphic (structurally identical).

The minimal DFAs are equivalent, then regular expressions **R** and **S** are equivalent.

2) Informal method to prove equivalence of two regular expressions

How can we determine whether two regular expressions denote the same language or not?

- Find a string that belongs to one regex and does not belong to the other regex to show that they are not equivalent.
- Faster, but based on hit and trail.
- Can only be used to prove inequality.

Example:

1) Are **R = a* (ba* ba*)*** and **S = a* (ba* b)* a*** equivalent?

Solution:

Find a string that belongs to one regex and does not belong to the other regex

bbaabb ∈ R

bbaabb ∉ S

Hence, R and S are not equivalent.

2) Are **R = a(a | b)*** and **S = (a(a | b))*** equivalent?

Solution:

Find a string that belongs to one regex and does not belong to the other regex

λ ∉ R

λ ∈ S

Hence, R and S are not equivalent.

3) Are **R = (0+λ) (11*0)* (1+λ)** and **S = (1+λ) (011*) (0+λ)** equivalent?

Solution:

Find a string that belongs to one regex and does not belong to the other regex

011 ∉ R

011 ∈ S

Hence, R and S are not equivalent.

Exercise:

I. Can you answer whether the two regular expressions are equivalent or not?

1) $(1+\lambda)\ (00^*1)^*\ 0^*$ and $(0+\lambda)\ (11^*0)1^*$

2) $0^*(10^*)^*$ and $(1^*0)^*1^*$

II. Construct Regular Expressions for the following languages:

i) L= {w | w is a binary string with even number of 0s}

ii) L= Set of all strings over the alphabet { a, b } that contain the substring ab.

iii) L= $\{a^nb^m \mid n+m \text{ is odd}\}$

iv) L= {Binary strings that have exactly one pair of 0's}

v) L= {w | w has a 1 at every odd position and |w| is odd}, Σ = {0, 1}

vi) L= Set of all strings over the alphabet { a, b } that do not end with 'ab'.

Solutions:

(i) ((1*01*01*)*)

(ii) (a + b)*ab(a + b)*

(iii) ((aa)*(bb)*b) + ((aa)*a(bb)*)

(iv) (1+01)* 00 (1+10)*

v) (1[01])*1 or 1((0|1)1)*

vi) $(\ a|b\ )^*\ (a|bb)\ |\ b$

Regex Construction:

Construct regular expression for the following statements over the alphabet Σ{a, b}:

Sl. No.	Statement	Regex
1	Strings ending with ab	(a|b)*ab
2	Strings containing ab	(a|b)*ab(a|b)*
3	Strings that start and end with the different symbol	a(a|b)*b | b(a|b)*a
4	String where the 3rd symbol from the left (i.e., from starting) is 'a'	(a|b)(a|b)a(a|b)*
5	String where at least one 'a' in the first three symbols and at least one 'b' in the last two Symbols.	((a(a|b)(a|b))| ((a|b)a(a|b))| ((a|b)(a|b)a)) (a|b)* (b(a|b)|(a|b)b)

6	Length of the string is greater than or equal to 2	(a|b)(a|b)(a|b)*
7	Length of the string is less than or equal to 2	λ|(a|b)|(a|b)(a|b)
8	Number of a's in the string should be at most 2	b*|b*ab*|b*ab*ab* **or** b*(λ|a)b*(λ|a)b*
9	L = { $n_a(w)$ mod 2 = 0}	(b*ab*ab*)* | b*
10	L = { $n_a(w)$ mod 2 = 1}	(b*ab*ab*)*b*ab*
11	L=($a^n b^m$ | n+m is even}	(aa)*(bb)* | (aa)*a(bb)*b
12	L=($a^n b^m$ | n,m ≥1 and n*m≥3}	**abbbb* | aaaa*b | aaa*bbb***
13	L=($a^{2n} b^{2m}$ | n,m≥1 }	aa(aa)* bb(bb)*
14	L=($a^n b^m$ | n≥4 and m≤3}	**aaaaa*** (λ|b) (λ|b) (λ|b)
15	L={$a^n b^m c^k$ | n+m is odd and k is even} **Note:** odd+even=odd and odd+odd=even	((aa)*a(bb)* | (aa)* (bb)*b) (cc)*
16	Strings over the alphabet a, b and where it does not have 2 consecutive a's and 2 consecutive b's	(b|λ) ab (a|λ) **or** (a|λ) ba (b|λ)
17	If the binary string is even then the length of the string is even, and if the binary string is odd then the length of the string is odd.	((0|1)(0|1))*(0|1)0 | ((0|1)(0|1))*1

Problem: Construct regular expression for the below language

`L = {Binary strings whose decimal equivalent is twice an odd number}`

Solution:

2* Odd no	Decimal value	Binary string
2*1	2	10
2*3	6	110
2*5	10	1010
2*7	14	1110
2*9	18	10010

The pattern what we can observe here is all the binary strings ends in 10

```
RE = (0+1)*(10)
```

References:

- Michael Sipser, Introduction to the Theory of Computation, 3rd Edition, Cengage Learning, June 27th 2012, ISBN: 9781285401065, 1285401069
- https://www.gatevidyalay.com/dfa-to-regular-expression-examples-automata/
- https://www3.ntu.edu.sg/home/ehchua/programming/howto/Regexe.html

Additional Info

Generalized NFA (GNFA)

In the theory of computation, a generalized nondeterministic finite automaton (GNFA), also known as an expression automaton or a generalized nondeterministic finite state machine, is a variation of a nondeterministic finite automaton (NFA) where **each transition is labeled with any regular expression**. The GNFA reads blocks of symbols from the input which constitute a string as defined by the regular expression on the transition.

There are several differences between a standard finite state machine and a generalized nondeterministic finite state machine.

- **A GNFA must have only one start state and one accept state, and these cannot be the same state, whereas an NFA or DFA both may have several accept states, and the start state can be an accept state.**
- A GNFA must have only one transition between any two states, whereas a NFA or DFA both allow for numerous transitions between states.
- In a GNFA, a state has a single transition to every state in the machine, although often it is a convention to ignore the transitions that are labelled with the empty set when drawing generalized nondeterministic finite state machines.

Generalized Transition Graph (GTG)

A generalized transition graph (GTG) is a transition graph whose **edges are labeled with regular expressions** or string of input alphabets rest part of the graph is same as the usual transition graph.

The label of any walk from initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expression are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Formal definition of generalized transition graph:

A generalized transition graph is defined by a 5-tuple, are as follows;

- Q : A finite set of states.
- Σ : A finite set of input symbols.
- S : A non-empty set of start states, $S \subseteq Q$
- F : A set of final or accepting states $F \subseteq Q$
- A finite set, δ of transitions, (directed edge labels) (u, s, v), where $u, v \in Q$ and s is a regular expression over Σ.

Generalized transition graphs are nondeterministic: Given a state and a [partially consumed] input, there may be more than 1 possible successor state.

Informally a generalized transition graph is a collection of three things are as follows;

1. GTG consists finite number of states, **at least one of which is start state** and **some (maybe none) final states.**

2. GTG consist finite set of input letters (Σ) from which input strings are formed.
3. Directed edges in GTG connecting some pair of states **labeled with regular expression**.

It may be noted that in GTG, the labels of transition edges are corresponding regular expressions.

Example:

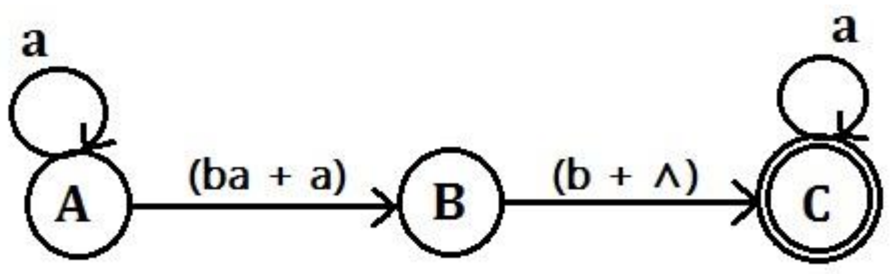


Figure 1 : Generalized Transition Graph (GTG) Example

This GTG accepts all strings without a double b. Note that the word containing the single letter b can take the free ride along the Λ-edge from start to middle, and then have letter b read to reach to the final state. The first edge should be labeled (ba + a) as in the figure above, not (ab + a).

Note: There is no difference between the Kleene star (*) closure for regular expressions and a loop in transition graphs, as illustrated in the following figure.

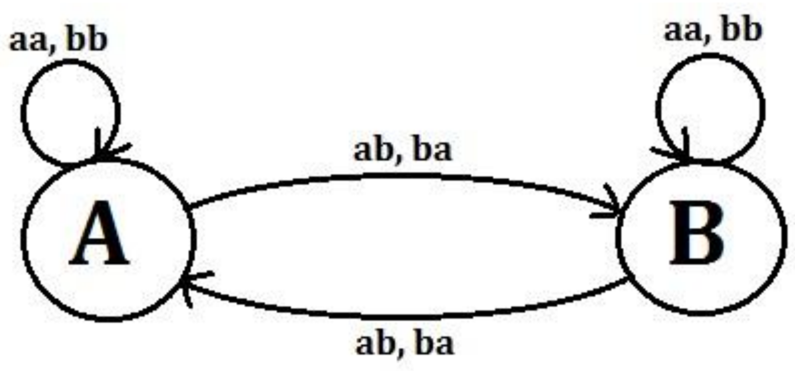
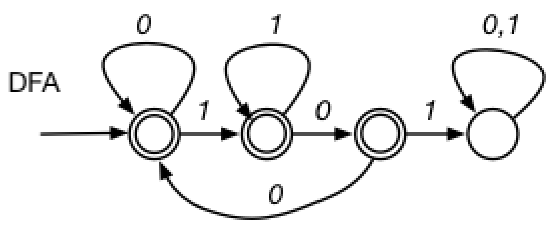


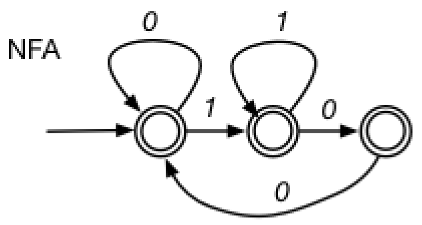
Figure 2 : Generalized Transition Graph (GTG) Example

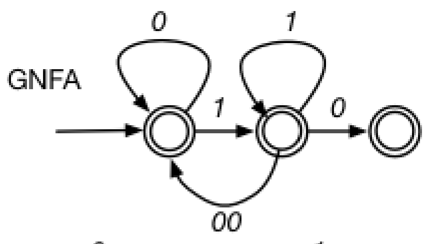
Consider the even-even language, defined over Σ = {a, b}. As discussed earlier that even-even language can be expressed by a regular expression ((aa+bb)* (ab+ba) (aa+bb)* (ab+ba))* The language even-even may be accepted by the following GTG.

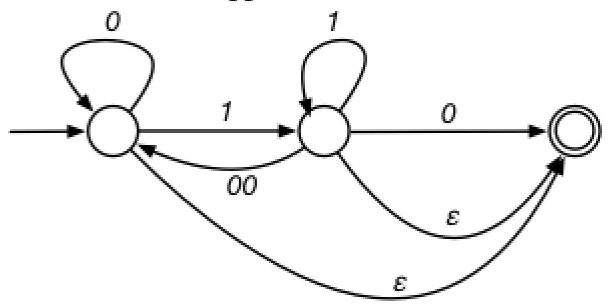
1) What is the regular expression of a string not containing 101 as a substring?

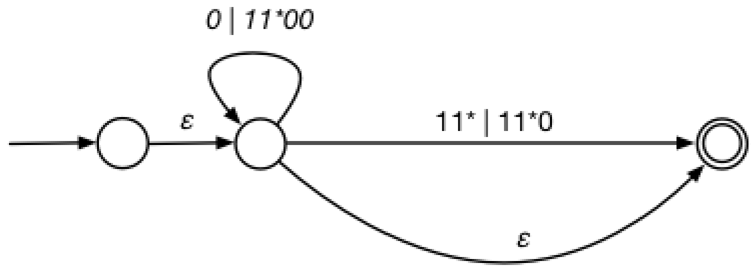
In the following construction I start with a DFA for $(101)^c$ which I then convert to a DFA and then into a sequence of GNFA's. The final GNFA yields the regular expression (which may not be the shortest possible).

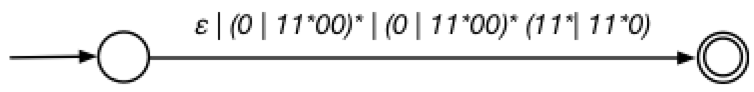








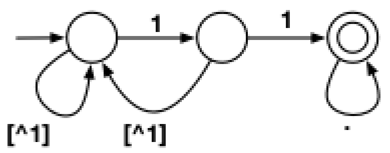




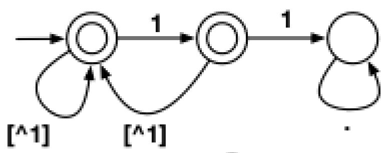
2) What is the regular expression for the string which doesn't contain 11 as a substring?

([^1] | (1[^1])*)* or simply **([^1] | 1[^1])***

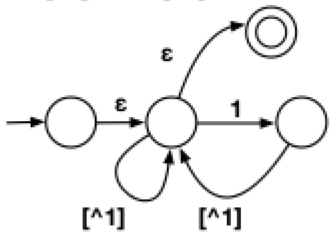
I arrived at this regular expression by first building a DFA for *L*= all strings that contain 11 as a substring. Then I built an DFA for the complement of *L*. Then there is a well known constriction for converting this to a regular expression.



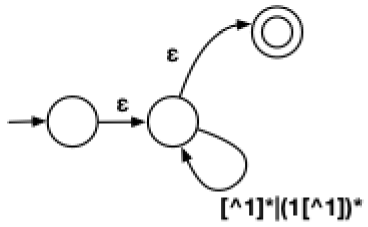
DFA for L = { .*11.*}



DFA for $\overline{L}$



NFA for $\overline{L}$



GNFA for $\overline{L}$

Greediness, Laziness and Backtracking for Repetition Operators

Greediness of Repetition Operators *, +, ?, {m,n}: The repetition operators are *greedy operators*, and by default grasp as many characters as possible for a match. For example, the regex xy{2,4} try to match for "xyyyy", then "xyyy", and then "xyy".

Lazy Quantifiers *?, +?, ??, {m,n}?, {m,}?, : You can put an extra ? after the repetition operators to curb its greediness (i.e., stop at the shortest match).

Backtracking: If a regex reaches a state where a match cannot be completed, it backtracks by unwinding one character from the greedy match. For example, if the regex z*zzz is matched against the string "zzzz", the z* first matches "zzzz"; unwinds to match "zzz"; unwinds to match "zz"; and finally unwinds to match "z", such that the rest of the patterns can find a match.

Possessive Quantifiers *+, ++, ?+, {m,n}+, {m,}+: You can put an extra + to the repetition operators to disable backtracking, even it may result in match failure. e.g, z++z will not match "zzzz". This feature might not be supported in some languages.