(FT-1) Language Generation :

Elements are the basic components.

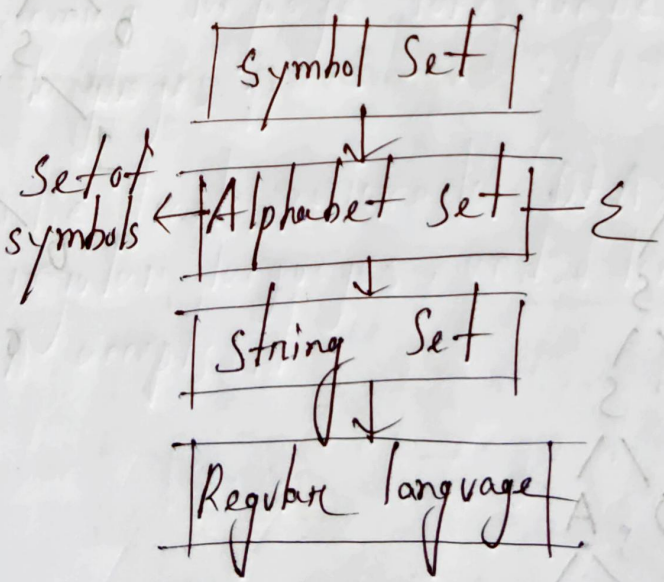
Two types : (I) Formal language → Used for specific application.

(II) Natural language

Formal / Language

Type → Used for specific application

Language Generation



A regular language is a language that can be expressed with a regular expression.

A language is a set of strings which are made up of characters from a specified alphabet.

Formal Language

** Regular expression are particular kind of formal grammar.

* Designed for specific application

Formal language consists of words whose letters are taken from an alphabet and are well formed according to a specific set of rules. The alphabet of a formal language consists of symbols or tokens that concatinate into strings of a language. Each string concatinated from the symbols is called a word.

rules

* Defined by means of formal grammar.

for example, regular grammar, context-free grammar.

** RE are used to parsed strings and other textual information that are known as Regular Language.

Regular language को produce गर्ने Regular grammar. Regular grammar लाई express गर्न Regular expression use गरिन्छ।

Regular expression :

*Regular expressions are notations to represent lexeme patterns for a token.

* Used to represent the language for lexical analyzer.

* They assist in finding the type of token that accounts for a particular lexeme.

⇒ The lexical analyzer scans and identifies only finite set of valid strings. It searches for pattern for defined by the language rules.

⇒ Programming language tokes can be described by regular languages.

Deterministic finite automata (DFA) recognise or accept regular languages, and a language is regular if a DFA accepts it.

R. Language

(strings)

{U . * +}

Regular expression operations:

1) * klene closure [a*] {ε, a, aa, aaa ----}
2) + positive closure [a+] {a, aa, aaa, ---}
3) . concatenation [a.b] {ab}
4) U union [a+b] {a+b}

Exercise:
Write R.E for following language over $\Sigma = \{a, b\}$

(1) Every string start with 'a' $a(a+b)^*$
(2) Every string start with 'ab' $ab(a+b)^*$
(3) Every string ends with 'aa' $(a+b)^*aa$
(4) Every string contain 'aba' as substring. $(a+b)^*aba(a+b)^*$
(5) Every string starts and ends with same symbol.

$(a(a+b)^*a \cdot b(a+b)^*b)$

Input Buffering

eol → end of line

end of file

		E	=	M	*	C	*	*	2	eof	

Symbol table

E	id
=	op

lexeme begin → pointer(2)

forward

*buffer → Short memory ↦ used in processor

A buffer contains data that is stored for a short amount of time, typically in RAM. The purpose of a buffer is to hold data right before it is used.

LEX & FLEX

lex (lex.l) → [lex Compiler] → lex.yy.c (translets in C file)

Source program

lex.yy.c → [C Compiler] → a.out → a file (its a lex analyzer)

input stream → [a.out] → sequence of token

FT (L-2)

Automation : (Automatic / Self controller)

TOC or TOA :

(I) Allows us to think systematically about computation.

(II) Allows us to design theoretical models for machine (abstract machine).

(III) Machine → Transfer to states

Finite Automata (It's a state machine)

↳ finite / limited number of states.

↳ Checking the string is valid or not. The string that is generated by Regular Language.

* Machine that takes input and gives output.
  ↳ language

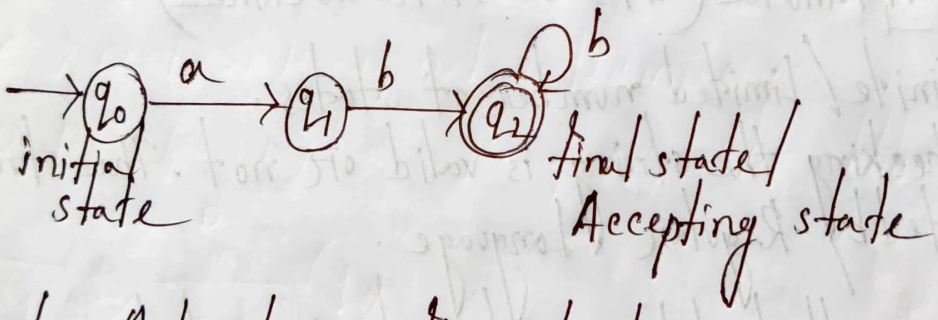
* Different ways we can define language.
* Define the language through graphical means.
* Define through transition table.
* Reorganize for language that is used to check weather a language accepts the string or not.

Finite Automata also consists of set of finite states and state of transitions from state to state that appears on input symbols chosen from an alphabet (Σ)

* How to define finite Automata by graphical means:

Transition Diagram
(to show the finite Automata)

R.E = ab^* ⟹ S_1 = abbb

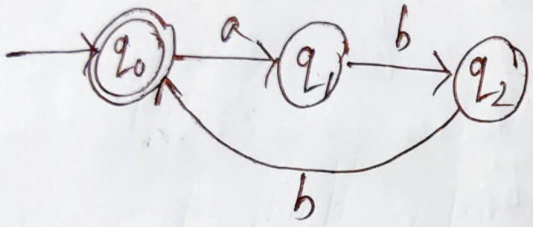


Finite Automata ⇒ five touple

$\{Q, q_0, F, \Sigma, \partial\}$

- Q → Set of state
- q_0 → initial state
- F → final state
- Σ → Alphabet set
- ∂ → transition number

$S_1 = aba$

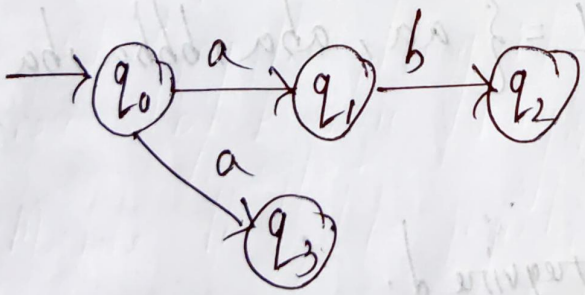


Type of Finite Automata — Fully Systematic — Next state is known

(I) Deterministic Finite Automata (DFA) (Something that is known)

(II) Nondeterministic Finite Automata (NFA)

↳ Don't know the next step.



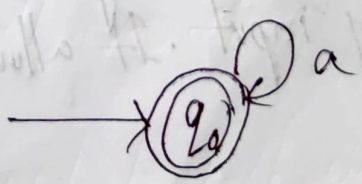
Examples of DFA: over $\Sigma = \{a\}$

(1) Contains any number of a

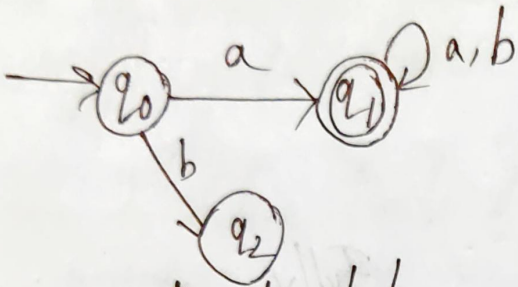
$R.E = a^*$

$L = \{\varepsilon, a, aa, aaa \cdots\}$

↳ empty string can't be in DFA



(ii) Starts with a , $\Sigma = \{a, b\}$, R.E = $a(a+b)^*$

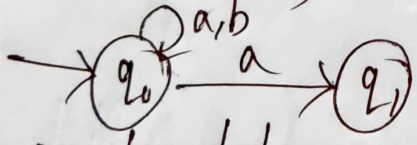


Death state

Examples of NFA :

Rule 1 : One letter can move to more than one state, next state is not determine.

R.E = $(a+b)^*a$ L = { aa , aba , bbba , ba }



* Dead state is not required.

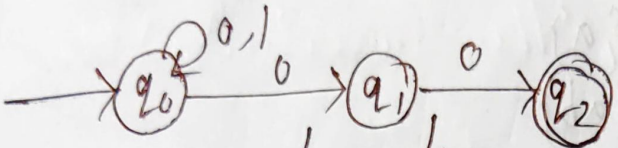
Rule 2 : Can go from one state to another without any input . It allows null transition / (Epsilon transition)

null ↓ doesn't exist

Epsilon transition ↓ Set is empty

Ques : The language $\{w \in \Sigma^* \mid w \text{ ends with } 00\}$ and $\Sigma = \{0,1\}$

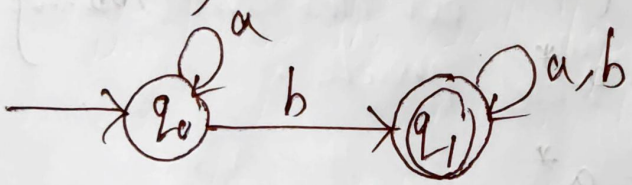
R.E = $(0+1)^* 00$



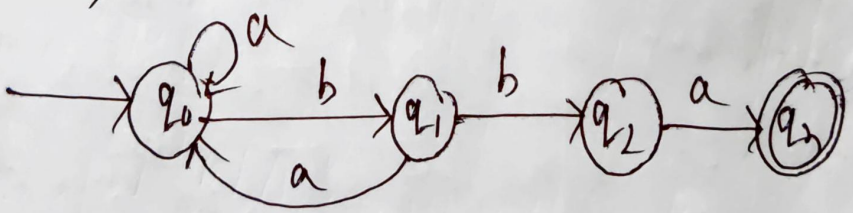
Same incoming transitions.

Convert RE to NFA

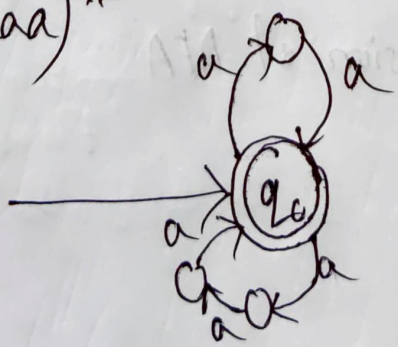
(I) $a^* b (a+b)^*$



(II) $(a+ba)^* ba$



(III) $(aa+aaa)^*$



FT (L-3)

~~R.E~~ Regular Expression to R.L.

R.E = a L = {a}

= a.b L = {ab}

= a+b L = {a, b}

= (a+b).b L = {ab, bb}

= a* L = {ε, a, aa, aaa ---}

= (a+b)* L = {ε, a, b, aa, ab, ba, bb ---}

= (a*)* L = a*

= (a+)* L = a*

FM

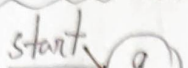
ε-NFA

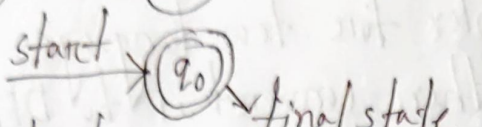
NFA

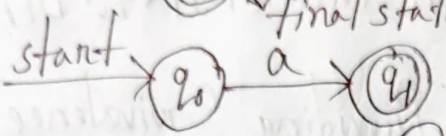
DFA

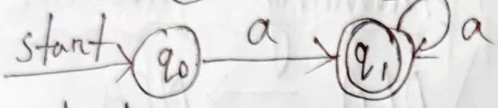
MDFA → optimize version of DFA

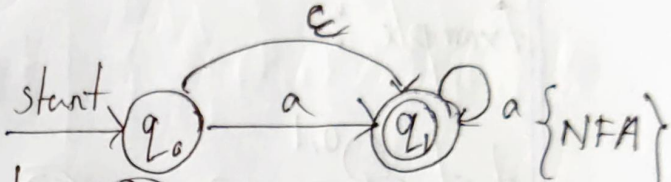
Primitive RE-FM

ϕ (null) 

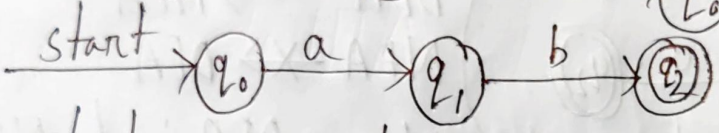
ε 

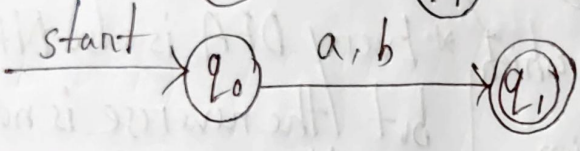
a 

a^+/a^* 

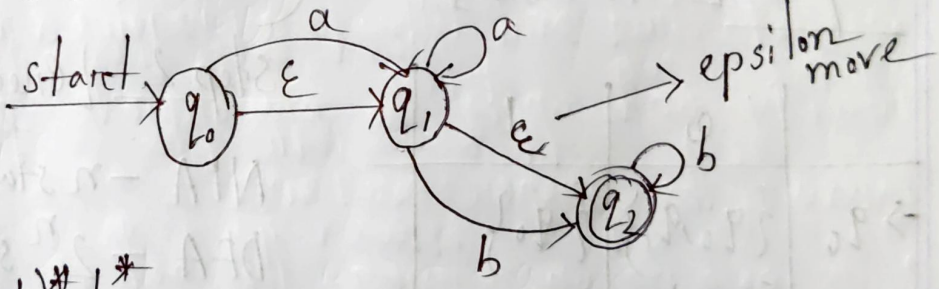


{NFA}

$a.b$ 

$a+b$ 

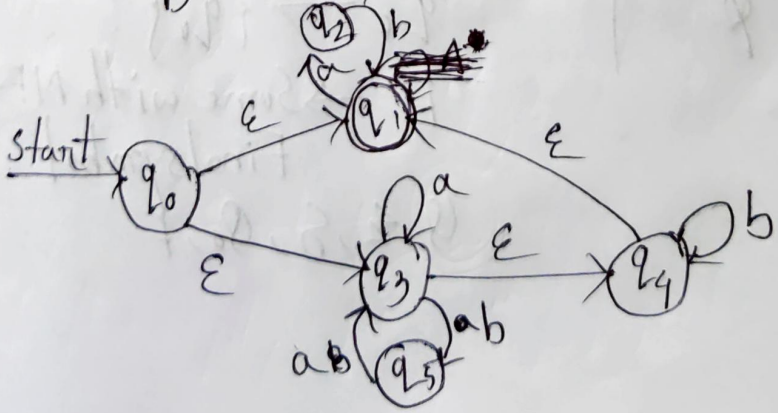
Few Complex R.E to FM

$a^* \cdot b^*$ 

~~L = {~~

$\frac{(a+b)^*}{A^*} + \frac{(a+ab)^* b^*}{B^*} \Rightarrow A^* + B^* b^*$

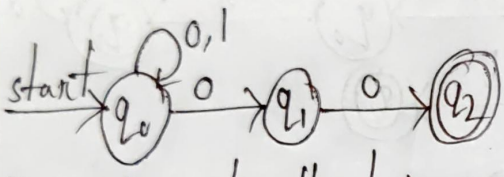
$S_1 = \varepsilon a b$



NFA to DFA
Subset Construction Method

* DFA creation is complex for few languages
* Create NFA first, then convert it to DFA

Example:



R.E - Accepts all strings which end with 00

→ transition table

step 1 [NFA table]

	0	1
→ q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	ϕ
* q_2	ϕ	ϕ

Providing Equivalence

DFA → NFA

NFA ↛ DFA

* Every DFA is also NFA, but the reverse is not possible.

step 2 [Create 5 tuples]

NFA - n states

DFA - 2^n states

Σ - Same

$q_0 \rightarrow \{q_0\}$

F → Same with NFA Final symbol.

$Q, \Sigma, \delta, q_0, F$

Step 3 [Symbol table]

DFA

		0	1
	$\emptyset$	$\emptyset$	$\emptyset$
→	$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
	$\{q_1\}$	$\{q_2\}$	$\emptyset$
*	$\{q_2\}$	$\emptyset$	$\emptyset$
	$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
*	$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
*	$\{q_1, q_2\}$	$\{q_2\}$	$\emptyset$
*	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

Step-4 [DFA - Transition Diagram]

Consider only the accessible states, Not all states are accessible from start.

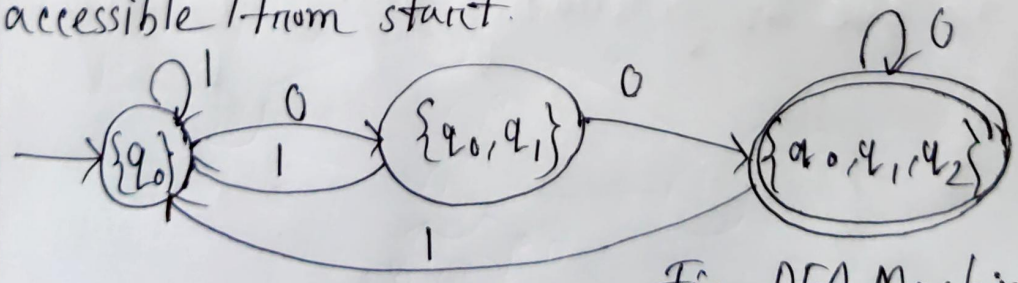


Fig - DFA Machine

Shortcut

{For more than 3 state}

	0	1
→ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
* $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

First & Follow

- Set of First and follow are required by parser to identify and apply the appropriate production rules at the proper position.
- In bottom-up parsing, the sets of first and follow helps in identifying when the reduction should be applied.

* Backtracking can be resolved in syntax Analysis phase by finding out first() set.

For example:

$P \rightarrow cAd$

$A \rightarrow bc \mid a$

inputstring "cad"

Backtracking can be avoided if the first() set of P and A are available.

How to find first() set elements:

→ If the first symbol of the production in R.H.S is:

(I) a terminal - include it directly in first() set

(II) a nonterminal - then call the first() function on that nonterminal and include those elements in the first set.

$S \rightarrow iEtST \mid a$ First(S) = {i, a}

$T \rightarrow eS \mid Null$ First(T) = {e, null}

$E \rightarrow b$ First(E) = {b}

Example:

$E \rightarrow TR$ — First(E) = {(, i}

$R \rightarrow +TR \mid \#$ — First(R) = {+, #}

$T \rightarrow FY$ — First(T) = {(, i}

$Y \rightarrow *FY \mid \#$ — First(Y) = {*, #}

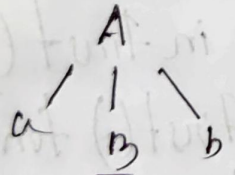
$F \rightarrow (E) \mid i$ — First(F) = {(, i}

* Why do we need follow?

$A \rightarrow aBb$

$B \rightarrow c \mid \underline{\varepsilon}$

input: "ab"



→ The issue here is that this can be applied only when compiler already knows that whether the character followed by B in production is same as current input character or not.

The use of follow() can make any non-terminal 'vanish' in case of generating any string wisely.

How do we find follow() set elements:

→ If any variable you have:

(a) Terminal - Then write it as it is.

(b) Non terminal - Then write the first element of its First() set.

(c) Last element of RHS: Then write Follow() of LHS

* Follow() set will never contain null
* Follow() of starting symbol will at least contain "$"

Example:

$S \rightarrow iEtST \mid a$ Follow(S) = {e, $}

$T \rightarrow eS \mid Null$ Follow(T) = {e, $}

$E \rightarrow b$ Follow(E) = {t}

$E \rightarrow TR$ Follow(E) = {), $}

$R \rightarrow +TR \mid Null$ Follow(R) = {), $}

$T \rightarrow FY$ Follow(T) = {+,), $}

$Y \rightarrow *FY \mid Null$ Follow(Y) = {*, +,), $}

$F \rightarrow (E) \mid i$ Follow(F) = {*, +, $,)}