Principles of Compiler Design

*) Compiler is an intermediate which is used to convert a high level language into low level language (or) machine understadable code (i.e., 0's and 1's).

Translators:

1) Compiler.
2) Assembler.
3) Interpreter.

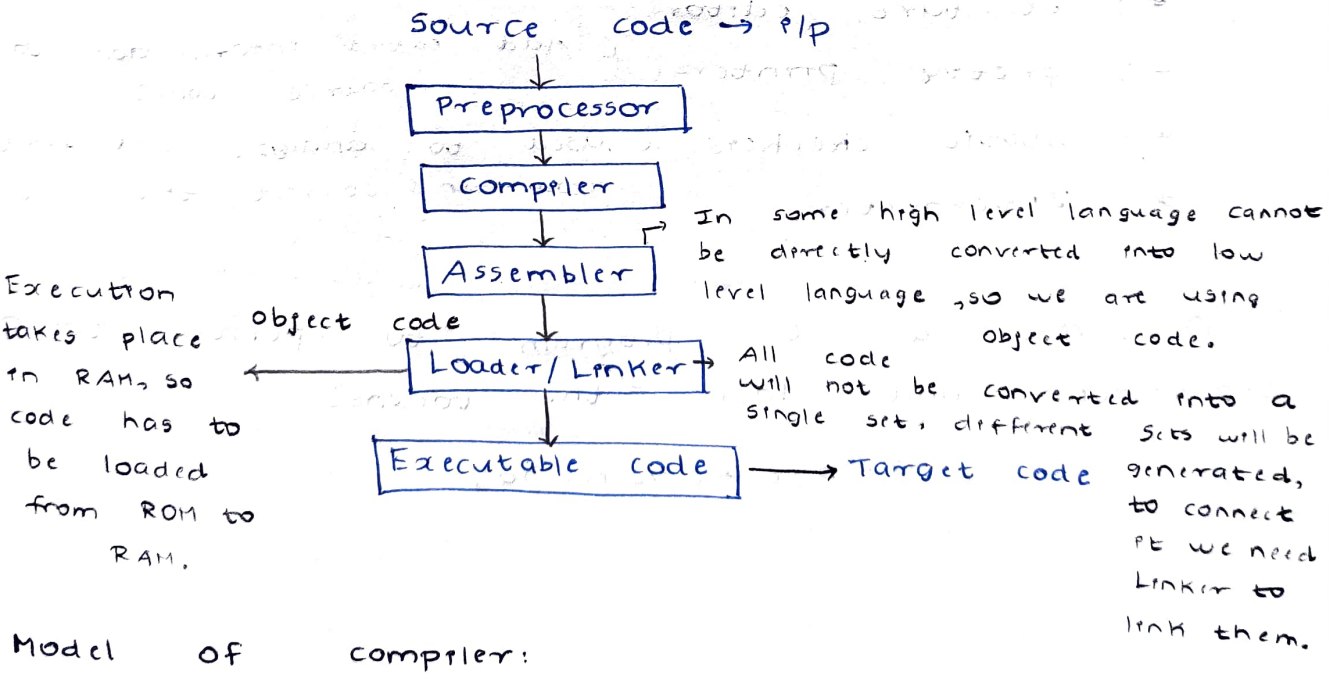
Source code → [Compiler] → Target code

Pneumonics is converted to low level language.
(cc = a+b, → ADD a,b)

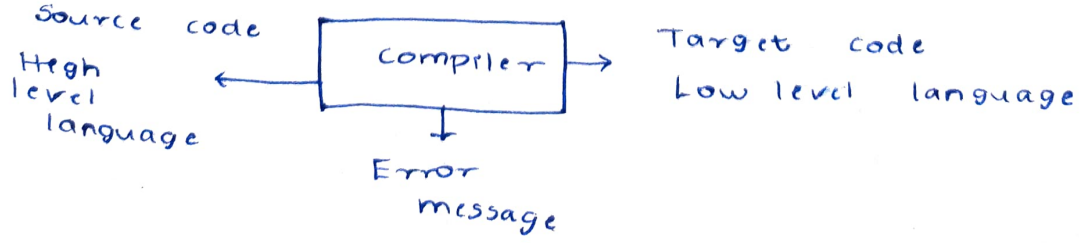
Compiler → converts into object code → generates an executable code. It also leads to additional memory

Interpreter → will not create a object code (or) generates an executable code. It does not requere an additional memory.

Language processing system:



Model of compiler:

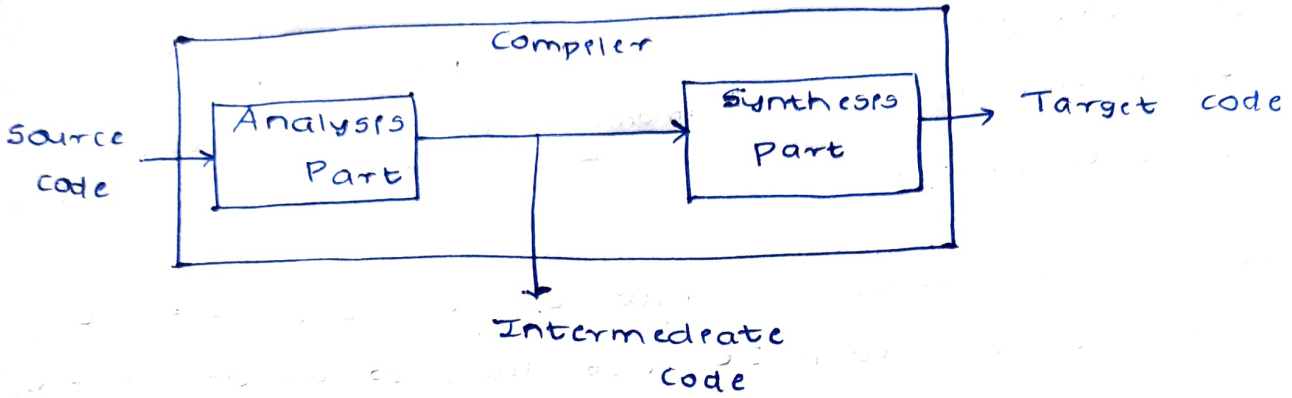


Process of compilation:

*.) Analysis Part.

*.) Synthesis Part.

Source program is read and broken down into small segments called as token in analysis part.



Token → identifiers, Keywords, Operators, seperators, constants.

Tools for manipulating source program and analyze the code:

*.) Structure editor. (uses command to build source program)

*.) pretty printers. (Add extra information to source code)

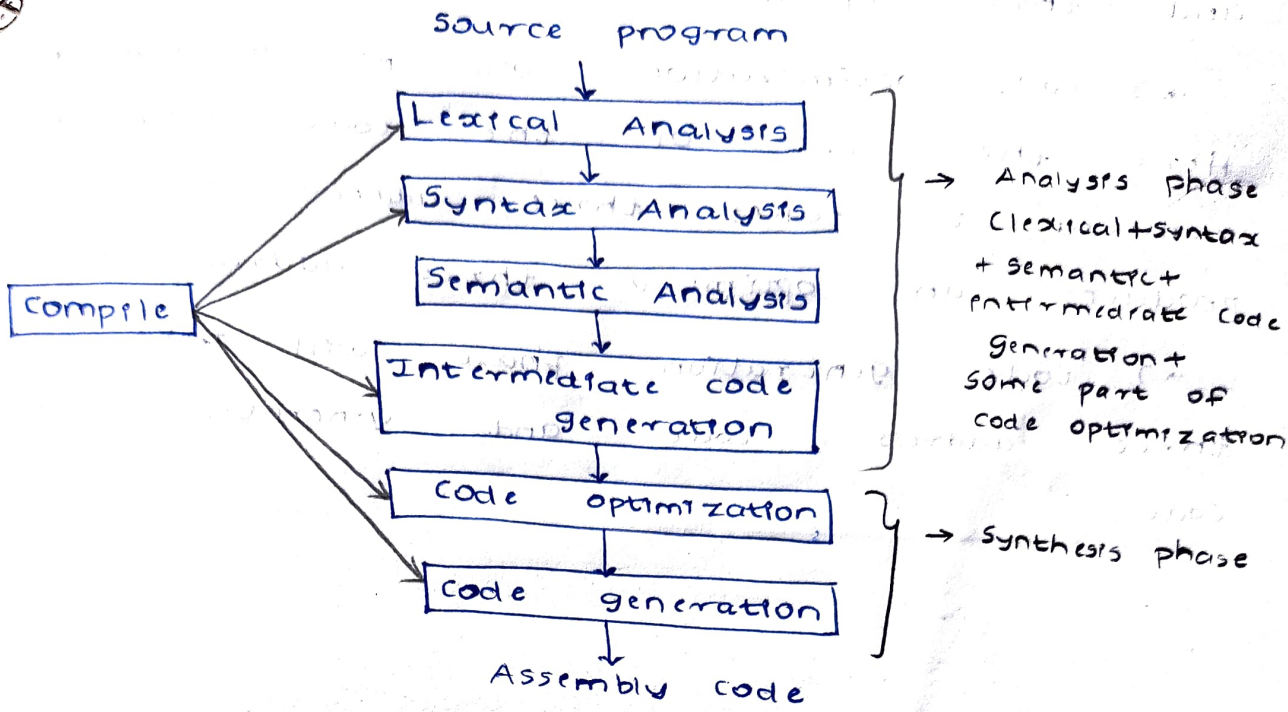
*.) static checkers. (used to analyze and find error & correct the code)

*.) Interpreter.

1) Write a C program to perform token seperation and label the tokens?

x = a+b/c*e%h-50;

phases of compilation:



Compile will mainly have two phases (i) Analysis phase (↦ similar to frontend) (ii) synthesis (↦ similar to backend). There are other two phases (i) Symbol Table (↦ contains details datatype of variable, memory allocated) (↦ used in lexical analysis) (ii) Error handling (used in all phases which accounts error handling mechanism).

*) Lexical analysis phase generates a set of tokens (Keywords, identifiers, constants, operators, delimiters). once tokens are generated, then it'll be moved to next stage. ↦ skipping of whitespaces, newlines.

*) Syntax analysis collects the tokens generated in the previous stage and create a tree called as syntax tree based on hierarchy.

*) semantic analysis checks the meaning of syntax (i.e., implicit or explicit type conversion).

*) Intermediate code generation goes through the tree and produces three address code.

$a + b \times c$
$t_1 = b \times c$
$t_2 = a + t_1$

} This will be the intermediate code generation.

Three address code → consists two operators and three operands.

*) code optimization will be optional phase. Here the size of the code will be reduced. It takes three address code and produces an optimised three address code.

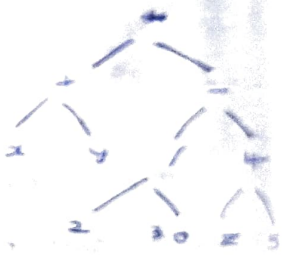
*) code generation phase will take three address code and generates assembly code.

Lexical Analysis:

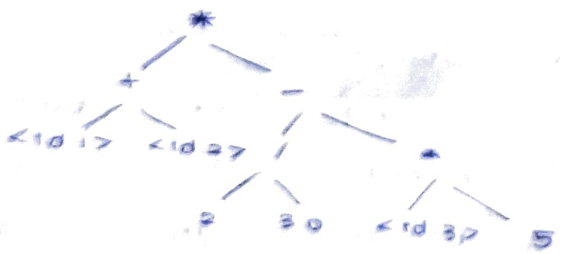
x+y * 2 / 30 - 2 * 5 where x & y are real.

lexical analysis

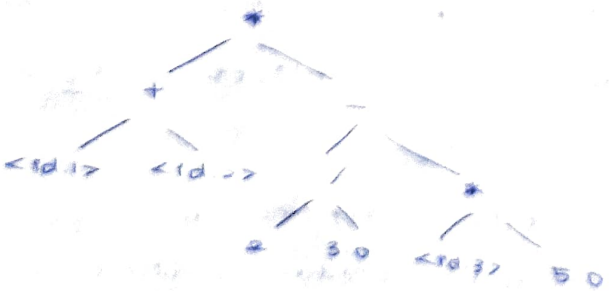
<id 1> <+> <id 2> <*> <2> </> <30> <-> <id 3> <*> <5>



Syntax analysis



Semantic analysis



Intermediate code generation

$t_1 = <id1> + <id2>$

$t_2 = float\ (2)$

$t_3 = t_2 / 3.0$

$t_4 = float\ (5)$

$t_5 = <id3> * t_4$

$t_6 = t_3 - t_5$

$t_7 = t_1 * t_6$

code optimization

$t_1 = <id1> + <id2>$

$t_2 = 2.0 / 3.0$

$t_3 = <id3> * 5.0$

$t_4 = t_2 - t_3$

$t_5 = t_1 * t_4$

Code generation

```
LDF   R1, id1
LDF   R2, id2
ADDF  R1, R1, R2
DIVF  R3, #2.0, #3.0
LDF   R4, id3
MULF  R4, R4, #5.0
SUBF  R3, R3, R4
MULF  R1, R1, R3
```

Reasons for seperation of lexical phase and syntax phase

*) Compiler simplicity. (or) simplicity in design.

*) Compiler portability.

*) Compiler efficiency.

Three important terms:

*) Token. → pair → token, symbol table entry.
   - token: abstract symbol which represents lexical unit.
   - symbol table entry: optional.

*) Pattern. → Rule or structure for framing a token.

*) Lexeme. → Sequence of characters that matches a pattern to form a token.

Error in lexical analysis:

*) Lexical analysis will show an error if any of the character that doesn't matches any of the pattern.

*) Error handler will try to delete a character, insert a character, replace characters (replace one character with other), transposing two adjacent character.

Panic error recovery mode:

(i) Delete a character.

(ii) Insert a character.

(iii) Replace a character with other character.

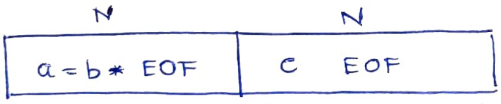
(iv) Transposing two adjacent characters.

Input Buffering:

↱ used to store characters while scanning.

*) In order to process a sequence of characters, an input buffer is used to store the sequence of characters. (One character mattu read pani token generate panna mudiyaathu, so we are using input buffer).

*) It is a buffer which is divided into two of size 4096 bytes.



It consists of two pointers

(i) Lexeme binary → current beginning of lexeme.

(ii) forward → to more forward.

*) For every character we have to check whether it is EOF. If EOB is read in first buffer, reload the second buffer and if End of Buffer (EOB) is read in second buffer, reload the first buffer.

*) It takes more time for checking this.

*) To reduce this time, we are using sentinels which uses the concept of EOF. only EOF can be checked.

String = "Kongu";

Prefix	∈, K, Ko, Kon, Kong, Kongu
proper prefix	K, Ko, Kon, Kong
Suffix	∈, u, gu, ngu, ongu, Kongu
Proper suffix	u, gu, ngu, ongu
Substring	continuous sequence (K, o, n, g, u, Ko, on, ng, gu, Kon, ong, ngu, Kong, ongu, Kongu)
Subsequence	characters may not be in sequence.

What will the prefix, proper prefix and substring when a string of length n is given?

Soln:

Prefix => n+1

Proper prefix => n-1

Substring => $\frac{n(n+1)}{2} + 1$ -> if we include empty substring

Suffix = n+1

subsequence = 2^n

Regular Expression: used to describe the pattern).

Regular expression has only three operators.

(i) closure (*)

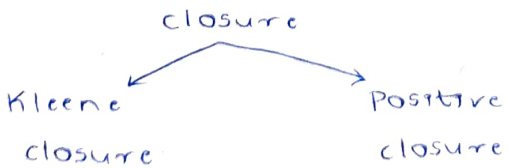
(ii) Concatenation (.)

(iii) Union (/ or +).

Priority,

* > . > / (or) +

*) closure is of two types.



*) String is a set of alphabets which is represented by Σ. (Sigma)

consider,

$\Sigma = \{a, b\}$

↳ Denotes create a string of length 0 from the above string.

$\Sigma^0 = \{\epsilon\}$ → since it is zero, nothing so epsilon will be used.

$\Sigma^1 = \{a\}$

$\Sigma^2 = \{ab, ba\}$

*.) $L = \{$collection of strings$\}$

$L = a^*$, $L = a^+$ (1 or more) ↳ Erlo renaalu raralaam but 'a' atleast one time raranum.

$L = a^*$ ← $L = \{\epsilon, a, aa, aaa, aaaa, \ldots\}$ (Kleene closure)

$L = a^+$ ← $L = \{a, aa, aaa, aaaa, \ldots\}$ (positive closure)

* → includes epsilon

+ → does not includes epsilon.

convention for naming a ramable => $l \cdot (l/d)^*$

↓ ↳ ramable (or) digit ↳ any no. of times

ramable +

~~Regular~~ ~~Expressions~~:

=

=

* → 0 or more

+ → 1 or more

? → 0 or 1

$L = a^*$

$L = \{\epsilon, a, aa, \ldots\}$

$L = L^0 \cup L^1 \cup L^2 \ldots$

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

$L^+ = \bigcup_{i=1}^{\infty} L^i$

$L^? = L^0 \cup L^1$

$L^* = L^+ + \epsilon$

$L^+ = L^+ - \epsilon$

*) If Regular Expression is taken as RE and set of strings that can be formed (or language) is taken as L.

(i) RE = {ε} , L = {ε}

(ii) RE = {a} , L = {a}

(iii) RE = {a/b} , L = {a, b}

(iv) RE = a.(a/b) , L = {aa, ab}

(v) RE = a.(a/b)* , L = {a, aa, aaa, ab, abbb, aaba....)

(vi) RE = (a/b).(a/b) , L = {aa, ab, ba, bb}

*) Given a set of all possible strings, find the Regular expression.

(i) L = set of all strings that starts with '0' over {0,1}.

L = {0, 000, 00, 01, 011, ... }

RE = 0.(0/1)*

(ii) L = set of all string having odd number of 1 over {0,1}

L = {1, 11, 111, ... }

RE = 0*.1.(0)*.(11)*.(0)*

(iii) L = set of all strings with 01 as substring {0,1}

RE = (0/1)*.0.1.(0/1)*

(iv) L = set of all strings that ends with bb over {a,b}

RE = (a/b)* bb

Regular expressions are recognized by finite automata.

Regular definition:

letter → [A-Z a-z]

digit → [0-9]

id → letter (letter/digit)*

digits → [0-9]+ (or) [0-9][0-9]*

float number → [0-9]+, [0-9]*

Finite Automata: (If n is given, the no. of states will be n+1)

*) Simple transition model that accepts string of Regular Expression.

*) Finite Automata
- Deterministic Finite Automata (DFA)
  - has more efficiency in processing
- Non-deterministic Finite Automata (NFA)
  - occupies less memory than DFA.

*) Finite Automata is denoted by M (model)

$$M = (Q, \Sigma, \delta, q_0, F)$$

where,

Q → set of all states of finite automata.

Σ → set of input alphabets.

δ → transition function.

q_0 → initial state of automata.

F → Final state of automata.

DFA => If a finite automata model has only one input on a transition to other state. Every symbols of finite automata should have only one transition.

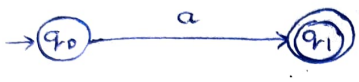
NFA => In a finite automata model, if an input symbol having more than one transition is to more than one state called as non-deterministic.

Regular expression defines a pattern for lexical analysis.

Example of DFA,

$L = \{a\}$

$RE = \{a\}$



Transition diagram

$M = (Q, \Sigma, \delta, q_0, F)$

$Q = \{q_0, q_1\}$

$\Sigma = \{a\}$

$\delta(q_0, a) = q_1$

$q_0 = q_0$

$F = q_1$

Set of all states	Input values: a
q_0	q_1
q_1	–

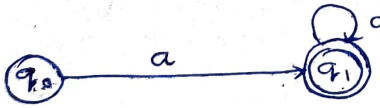
○ → initial state

◎ → Final state

(ii) $L = \{a, aa, aaa, \ldots\}$

$RE = a^+$

Whenever an finite automata model is drawn, it should be drawn for minimal string.

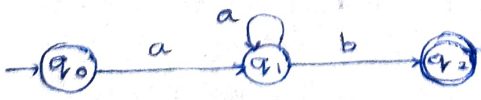


$RE = a^*$

(iii) $L = \{ab\}$

$RE = ab$

(iii) $L = \{ab, aab, aaab, aaaab, \ldots\}$, $RE = a^+b$



Set of all states \ Input values	a	b
q_0	q_1	–
q_1	q_1	q_2
q_2	–	–

1) L = set of all strings with odd number of 1's over {0,1}

L = {1, 111, 11111, ... }

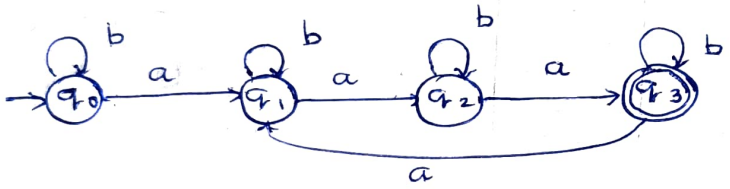
RE = $0^*.1.(0)^*.((11)^*.(0)^*$

	0	1
q_0	q_0	q_1
q_1	q_1	q_0

2) L = set of all strings over {a,b} that accepts string with number of 'a' is divisible by 3.

Soln:

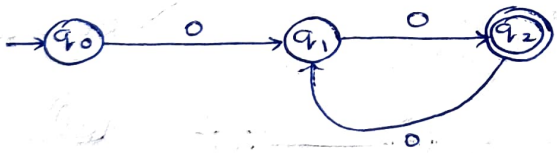
RE = $b^*(aaa)^+$ $b^*(ab^*ab^*a)^+b^*$



3) L = {00, 0000, 000000, ... }

Soln:

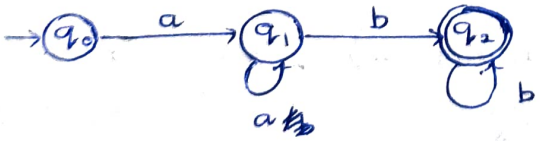
RE = $(00)^+$



4) RE = $(a/b)^*ab$

Soln:

L = {ab, aab, aabb, ...}



5) L = {01, 00111, 000100, ... }

Soln:

RE = $(00)^*(01)^*$

*) NFA can have ε as input (i.e., It can make transition from one state to another state without reading any input).

*) NFA occupies less memory than DFA.

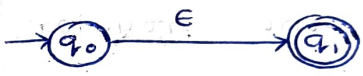
*) DFA is more efficient in processing. So, NFA should be converted into DFA while processing.

*) DFA cannot have ε as input.

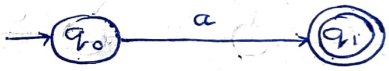
Rules for converting Regular expression into ε-NFA:

Regular Expression ε-NFA

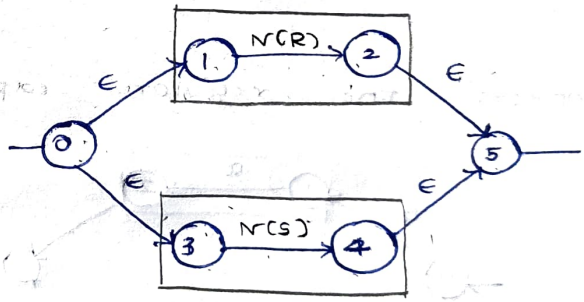
1) ε



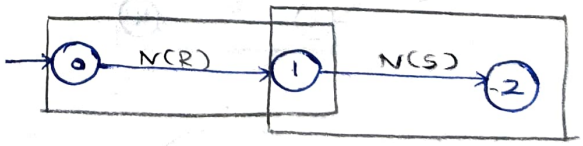
2) a



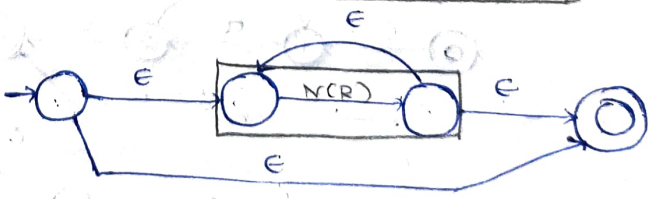
3) RS → R/S
(or)
R+S
a/b (or) a+b



4) R.S
a.b



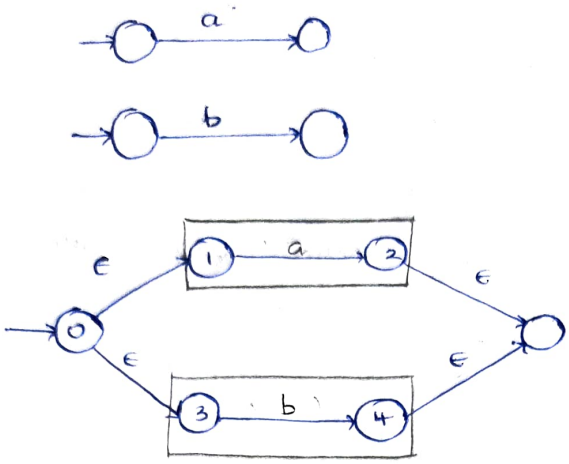
5) R* a*



The process of converting a regular expression into ε-NFA is called as Thomson's construction method.

1) Convert the regular expression a/b to ϵ-NFA.

Soln:

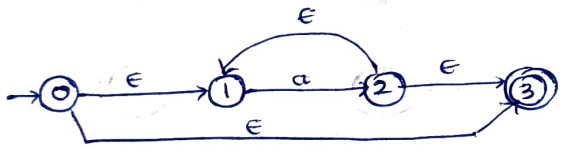


2) Convert the regular expression a.b to ϵ-NFA.

Soln:

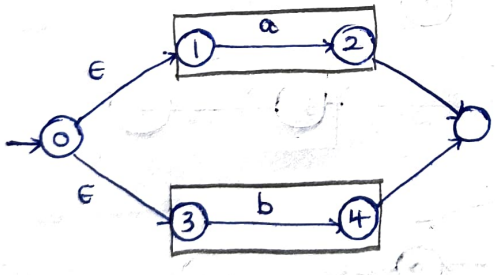
3) Convert the regular expression a* to ϵ-NFA.

Soln:

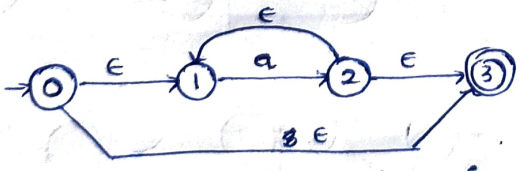


4) Convert the regular expression (a/b)* to ϵ-NFA.

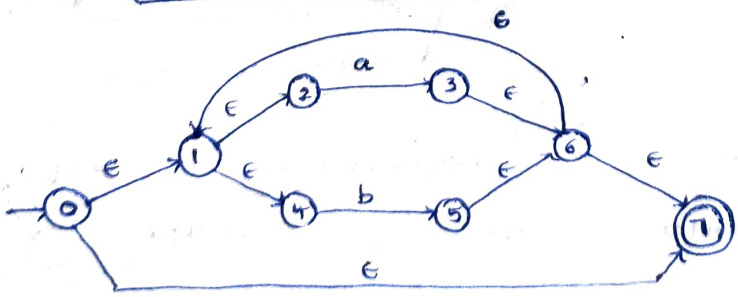
Soln:



⇒ (a/b)



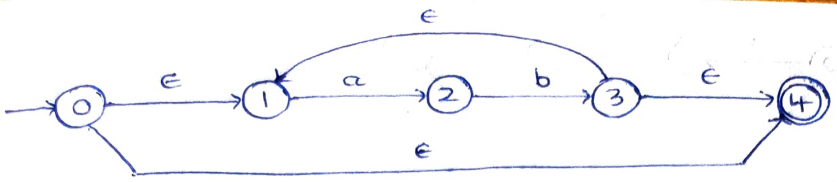
⇒ a*



⇒ (a/b)*

5) Convert the regular expression (a.b)* ~~for~~ to ϵ-NFA.

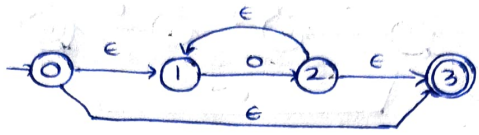
Soln:



6) Convert the regular expression $01/0^*$ into ε-NFA.

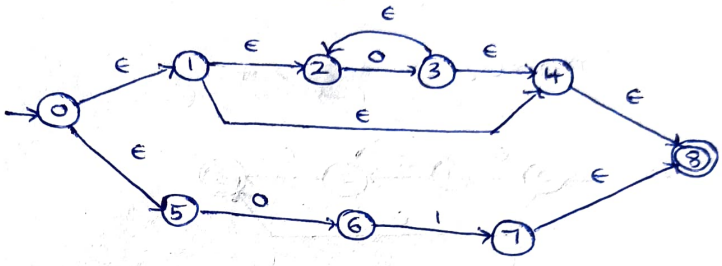
soln:

0^* =>



01 =>

$01/0^*$ =>



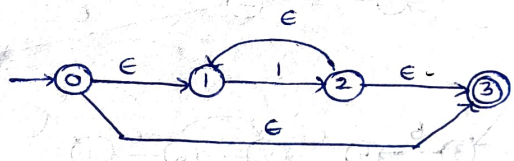
7) Convert the regular expression into ε-NFA

(i) $1^*/0^*$ (ii) $1^*.0^*$

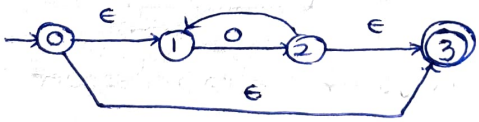
soln:

(i) $1^*/0^*$

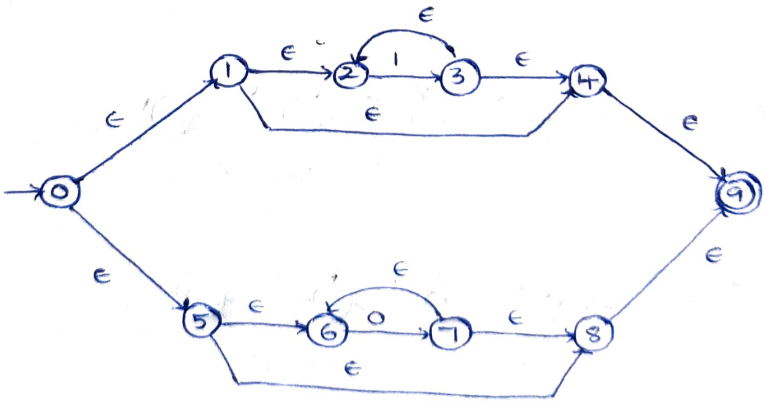
1^* =>



0^* =>

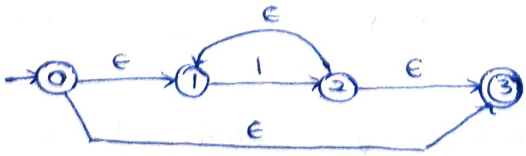


$1^*/0^*$ =>

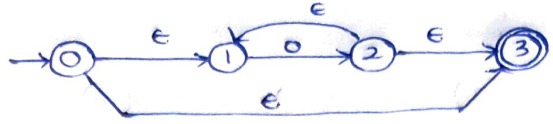


(ii) $1^*.0^*$

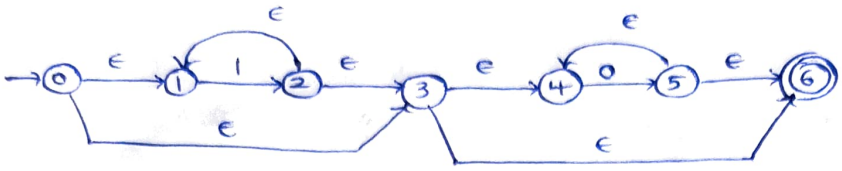
1^* =>



0^* =>



$1^*.0^*$ =>

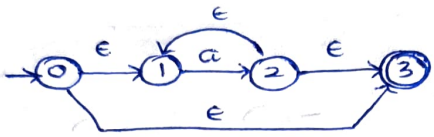


8) convert the regular expression $a+b.a^*$ into ϵ-NFA.

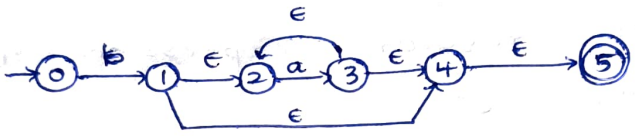
Soln:

Prority => * > . > +

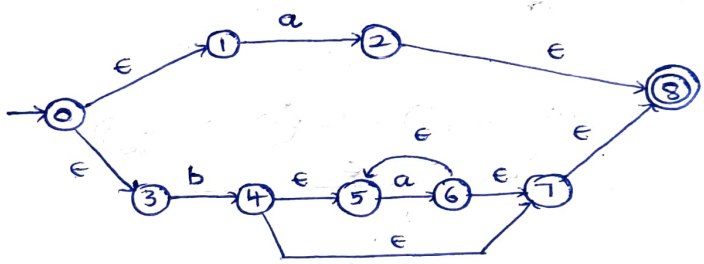
a^* =>



$b.a^*$ =>



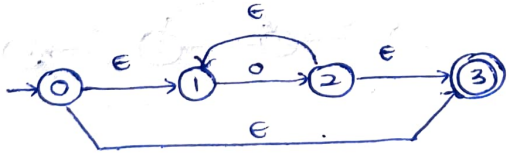
$a+b.a^*$ =>



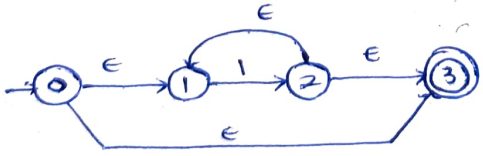
9) convert the regular expression $1/0^*/1^*$ into ϵ-NFA.

Soln:

0^* =>



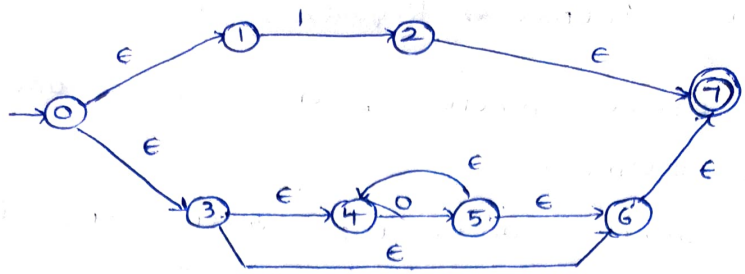
1^* =>



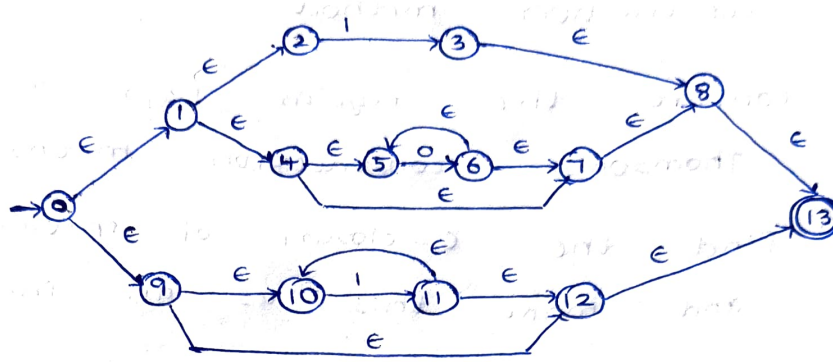
1 =>

$1/0^*/1^*$ => Associativity from Left to Right.

1/0* =>



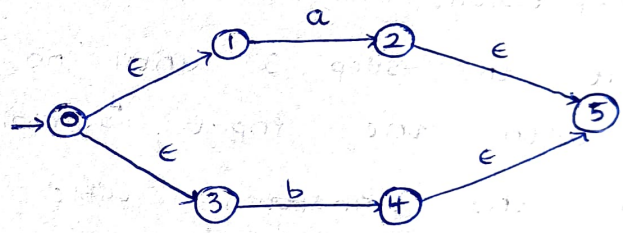
1/0*/1* =>



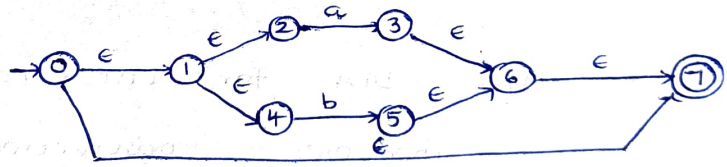
10) convert the following regular expression (a/b)* abb into ϵ-NFA.

Soln:

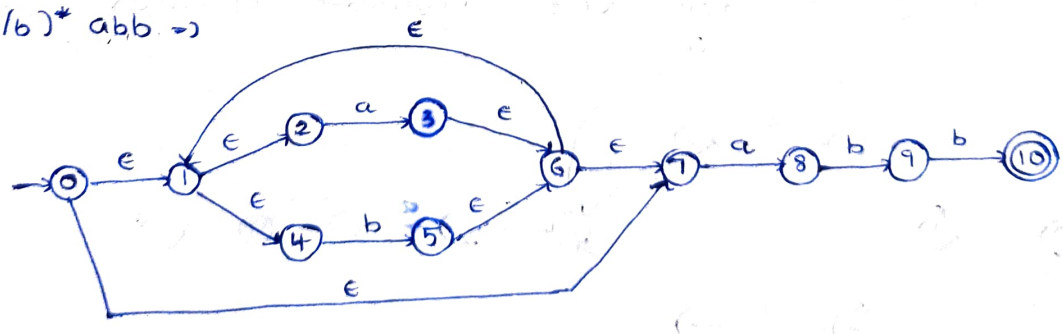
(a/b) =>



(a/b)* =>



(a/b)* abb =>



Regular Expression to NFA to DFA :

Two ways to convert regular expression into DFA is :

(1) Subset construction Method => Regular expression to ϵ-NFA - then to DFA.

(ii) Direct Method ⇒ Regular Expression - DFA

*.) In direct method, the DFA obtained is minimized (optimal).

*.) In subset construction method, the DFA obtained must undergo minimization.

Subset construction method:

1) Convert the regular expression to ϵ-NFA using Thomson's construction method.

2) Find the ϵ-closure of initial state of ϵ-NFA and make this as an initial state of DFA.

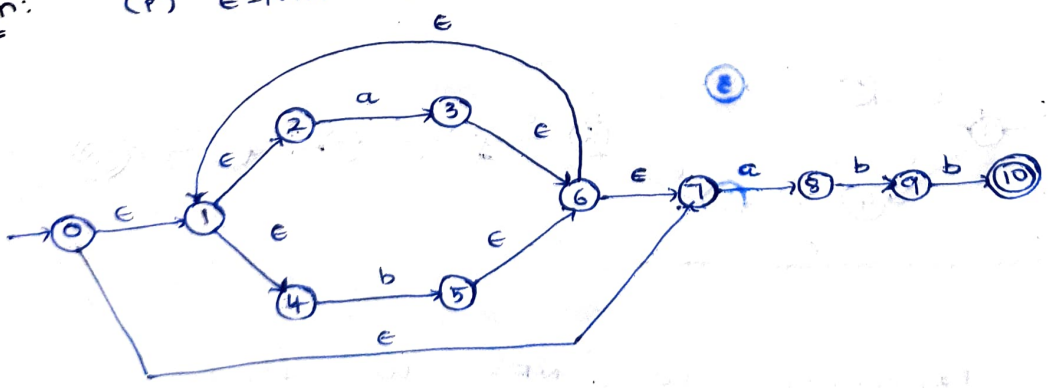
3) Find the moves on the initial state of DFA with all the input symbols in the regular expression.

4) Repeat the step 3 until no new state is obtained with the input symbols.

5) Draw the transition table and the transition diagram which is the required DFA.

1) construct the DFA for the regular expression (a/b)* abb using Thomson's construction method.

Soln: (i) ϵ-NFA for the regular expression



(ii) ϵ-closure of initial state

Set of all states reached by reading ϵ ⇒ ϵ-closure

ϵ-closure (0) = { 0, 1, 2, 4, 7 }

Make this as initial state of DFA,

ϵ-closure (0) = { 0, 1, 2, 4, 7 } → A

(All) → ϵ-closure la irukura values la ethu a, vth input ah eduthu enna state ku pothuna eluthanum,

*) Move (A, a) = { 3, 8 }

↳ Regular expression la irukka ella input state kum podanum

ϵ-closure (Move (A, a)) = ϵ-closure ({ 3, 8 })

= ϵ-closure (3) ∪ ϵ-closure (8)

= { 3, 6, 7, 1, 2, 4 } ∪ { 8 }

= { 1, 2, 3, 4, 6, 7, 8 } → compare this with old state A, which is different so assign it as a new state B.

B = { 1, 2, 3, 4, 6, 7, 8 }

Move (A, b) = { 5 } = C

ϵ-closure (Move (A, b)) = ϵ-closure (5)

= { 5, 6, 7, 1, 2, 4 } → C

*) Move (B, a) = { 3, 8 }

∴ = { 1, 2, 3, 4, 6, 7, 8 } → B

Move (B, b) = { 5, 9 }

ϵ-closure (Move (B, b)) = ϵ-closure ({ 5, 9 })

= ϵ-closure (5) ∪ ϵ-closure (9)

= { 5, 6, 7, 1, 2, 4 } ∪ { 9 }

= { 1, 2, 4, 5, 6, 7, 9 } → D

*) Move (C, a) = { 3, 8 }

ϵ-closure (Move (C, a)) = ϵ-closure ({ 3, 8 })

= ϵ-closure (3) ∪ ϵ-closure (8)

= { 3, 6, 7, 1, 2, 4 } ∪ { 8 }

= { 1, 2, 3, 4, 6, 7, 8 } → B

Move (C, b) = { 5 }

ϵ-closure (Move (C, b)) = ϵ-closure (5)

= { 5, 6, 7, 1, 2, 4 } → C

*.) Move (D, a) = {3, 8}

ϵ-closure({3,8}) = {1, 2, 3, 4, 6, 7, 8}

Move (D, b) = {5, 10}

ϵ-closure (Move (D, b)) = ϵ-closure({5, 10})

= ϵ-closure(5) ∪ ϵ-closure(10)

= {5, 6, 7, 1, 2, 4} ∪ {10}

= {1, 2, 4, 5, 6, 7, 10} → E

*.) Move (E, a) = {3, 8}

ϵ-closure({3,8}) = {1, 2, 3, 4, 6, 7, 8}

Move (E, b) = {5}

ϵ-closure({5}) = {5, 6, 7, 1, 2, 4}

Move(A, a) → B, Move (A, b) → C, Move (B, a) → B, Move (B, b) → D,

Move(C, a) → B, Move (C, b) → C, Move (D, a) → B, Move (D, b) → E,

Move (E, a) → B, Move (E, b) → C

Transition Table

(iv)

	input	symbol
states	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

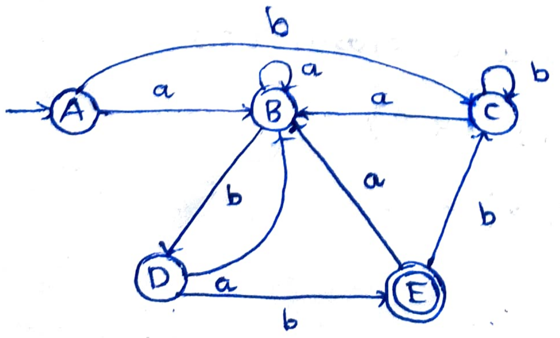
Initial state → A

Final state → E

ϵ-NFA last value en state la iruko, athu final state, if it present in more than one, all will be final state

Transition Diagram

(v)



2) Construct the DFA for the regular expression

i) $((\epsilon/a).b^*)^*$ (ii) $a^*/b^*.\ a/b.$

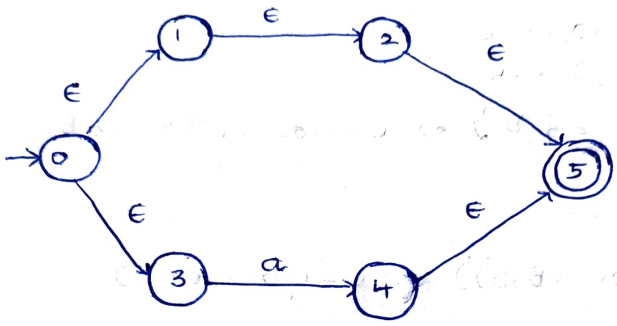
(i) $((\epsilon/a).b^*)^*$

Step 1: ϵ-NFA for the regular expression.

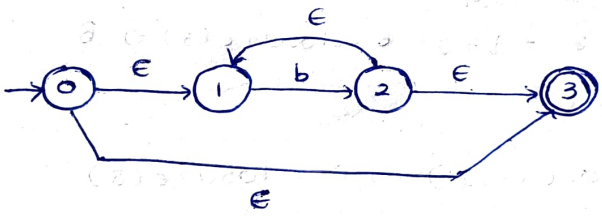
ϵ 

a 

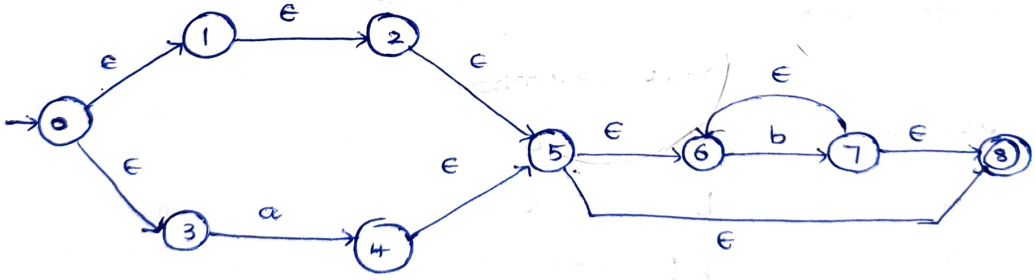
(ϵ/a)



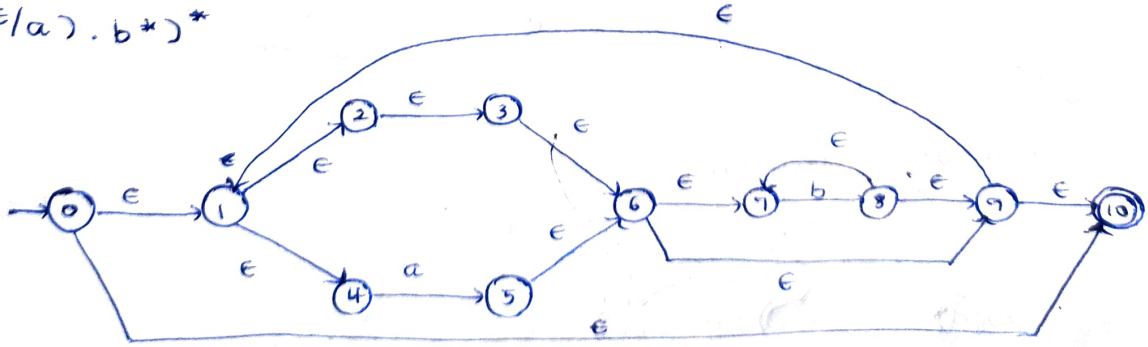
b^*



$(\epsilon/a).b^*$ ⇒



$((\epsilon/a).b^*)^*$



Step 2: ϵ-closure of initial state

ϵ-closure(0) = {0, 1, 2, 3, 6, 7, 4, 9, 10}

Make this as initial state of DFA.

ϵ-closure(0) = {0, 1, 2, 3, 4, 6, 7, 9, 10} → A

Step 3:

Move (A, a) = {5}

*) ϵ-closure (Move (A, a)) => ϵ-closure (5)

= {5, 6, 7, 9, 10, 1, 2, 3, 4} → B

Move (A, b) = {8}

*) ϵ-closure (Move (A, b)) => ϵ-closure (8)

= {8, 7, 9, 10, 1, 2, 3, 6, 4} → C

Move (B, a) = {5} => ϵ-closure (5) => B

Move (B, b) = {8}

*) ϵ-closure (Move (B, b)) => ϵ-closure (8)

= {8, 7, 9, 10, 1, 2, 3, 6, 4} → C

Move (C, a) = {5} = ϵ-closure (5) => B

Move (C, b) = {8}

*) ϵ-closure (Move (C, b)) => ϵ-closure (8)

= {8, 7, 9, 10, 1, 2, 3, 6, 4} → C

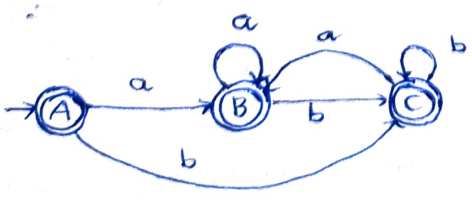
Step 4:

Transition table

	Input symbol	
States	a	b
A	B	C
B	B	C
C	B	C

Step 5:

Transition diagram



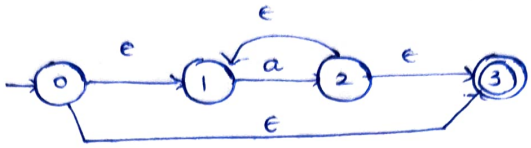
(rr) $a^*/b^*.a/b$

soln:

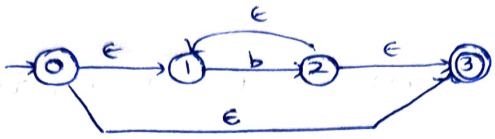
$* > . > + (or) /$

Step 1:

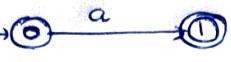
a^* =>



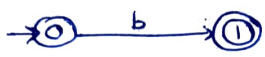
b^* =>



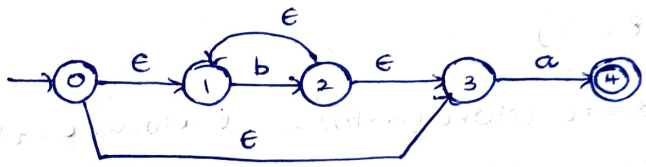
a =>



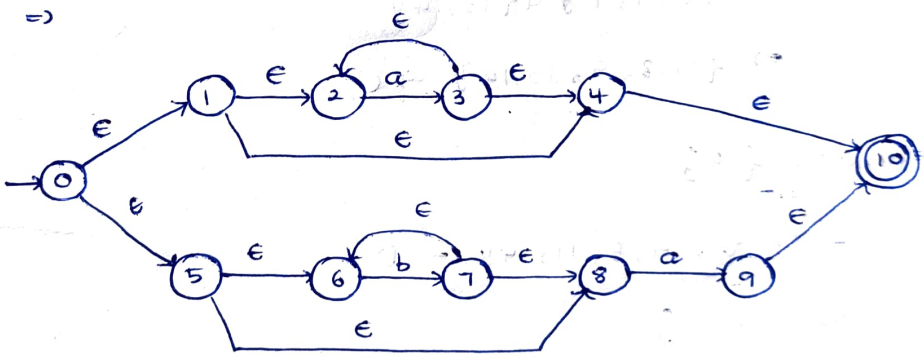
b =>



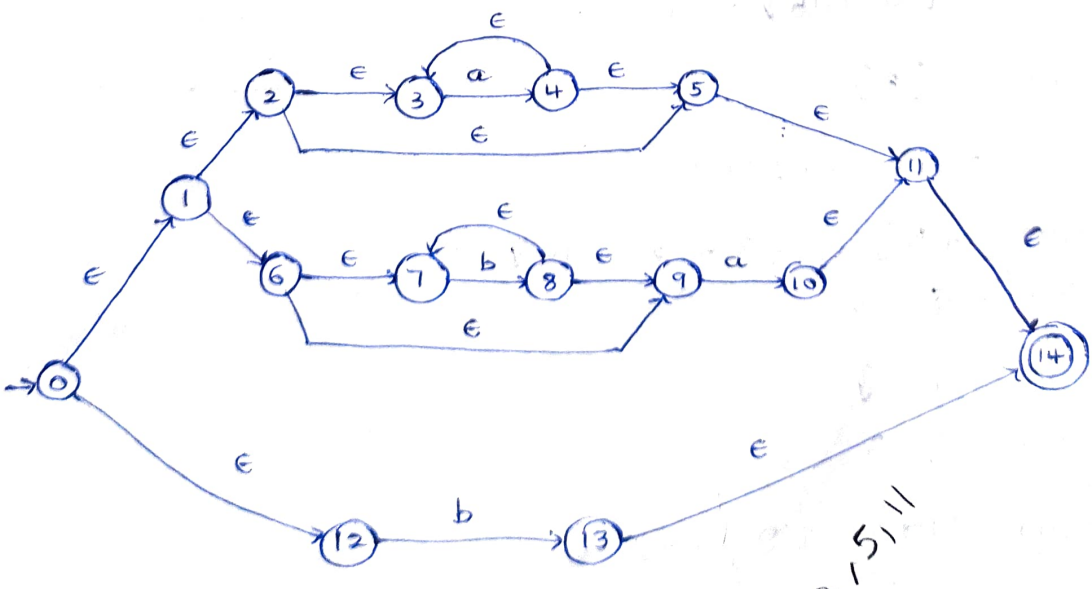
$b^*.a$ =>



a^*/b^*a =>



$a^*/b^*a/b$ =>



0 1 1 2 3 1 5 1 1

Step 2:

ϵ-closure (0) = {0, 1, 2, 3, 5, 11, 14, 6, 7, 9, 12}

A = {0, 1, 2, 3, 5, 6, 7, 9, 11, 12, 14}

Step 3:

Move (A, a) = {4, 10}

*.) ϵ-closure (Move (A, a)) => ϵ-closure ({4, 10})

=> ϵ-closure (4) $\cup$ ϵ-closure (10)

=> {3, 4, 5, 11, 14} $\cup$ {10, 11, 14}

=> {3, 4, 5, 10, 11, 14} → B

Move (A, b) = {8, 13}

*.) ϵ-closure (Move (A, b)) => ϵ-closure ({8, 13})

=> ϵ-closure (8) $\cup$ ϵ-closure (13)

=> {8, 7, 9} $\cup$ {13, 14}

=> {7, 8, 9, 13, 14} → C

Move (B, a) = {4}

=> {3, 4, 5, 11, 14} → D

Move (B, b) = { }

Move (C, a) = {10}

= {10, 11, 14} → E

Move (C, b) = {8}

= {8, 7, 9} → F

Move (D, a) = {4} => {3, 4, 5, 11, 14} → D

Move (D, b) = { }

Move (E, a) = { }

Move (E, b) = { }

Move (F, a) = {10} => {10, 11, 14} → E

Move (F, b) = {8} => {8, 7, 9} → F

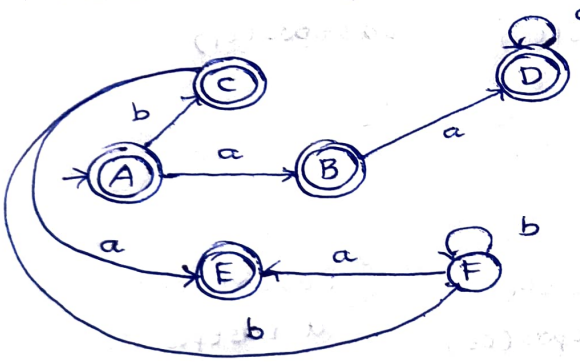
Step 4:

Transition table

	Input symbol	
states	a	b
A	B	C
B	D	-
C	E	F
D	D	-
E	-	-
F	E	F

Step 5:

Transition diagram



Initial state of DFA - First position of root node of Syntax tree.

Regular Expression to DFA - Direct method:

Step 1: Place an unique end marker "#" for the regular expression 'R'. R# is called as augmented grammar.

Step 2: Any DFA state having a transition to "#" must be the final state of DFA.

Step 3: construct a syntax tree for the augmented grammar. The nodes of the syntax tree will be the operators of the regular expression. The leaves may be alphabets or "ε"

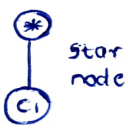
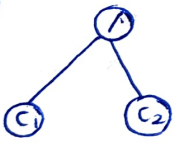
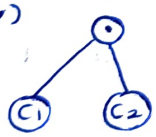
Step 4: (1) OR nodes are represented as "/".

(ii) cat nodes will be represented with '.'

(iii) star nodes will be represented with '*'

/ & . will have two childs. * have one child

Step 5: compute the Nullable, ~~Parent~~ first position, last position for the nodes and leaves of the Syntax tree using the following table. All leaves nodes except 'ε' will be given positions (i) which starts from 1

node	Nullable	firstpos	lastpos
(i) leaf is labelled with 'ε'	True	ϕ	ϕ
(ii) leaf is labelled with some position {i}	False	{i}	{i}
(iii)  star node	True	firstpos(c_1)	Lastpos(c_1)
(iv) 	~~False~~ Nullable(c_1) ∪ Nullable(c_2)	firstpos(c_1) ∪ firstpos(c_2)	Lastpos(c_1) ∪ Lastpos(c_2)
(v) 	~~False~~ Nullable(c_1) && Nullable(c_2)	if nullable(c_1) then firstpos(c_1) ∪ firstpos(c_2) else firstpos(c_1)	if nullable(c_2) then lastpos(c_1) ∪ lastpos(c_2) else lastpos(c_2)

Step 6: (i) Compute the follow position for * and cat(.) node.

(ii) If 'n' is a star node then,

lastpos(n) ← firstpos(n)

(iii) If 'n' is a cat node then,

lastpos(c1) ← firstpos(c2)

Step 7: Compute the DFA transition with the initial state as the first position of the rootnode of the syntax tree.

Step 8: The obtained DFA is the minimised DFA.

1) Construct a DFA for RE $(a/b)^*abb$ using direct method.

Soln:

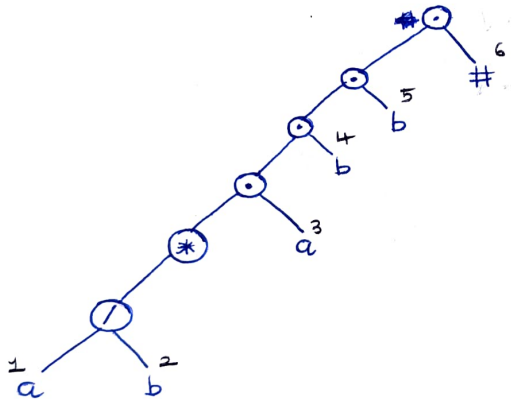
(i) Add an #(unique marker) to the end of the regular expression.

Augumented grammar

$(a/b)^*.a.b.b.\#$

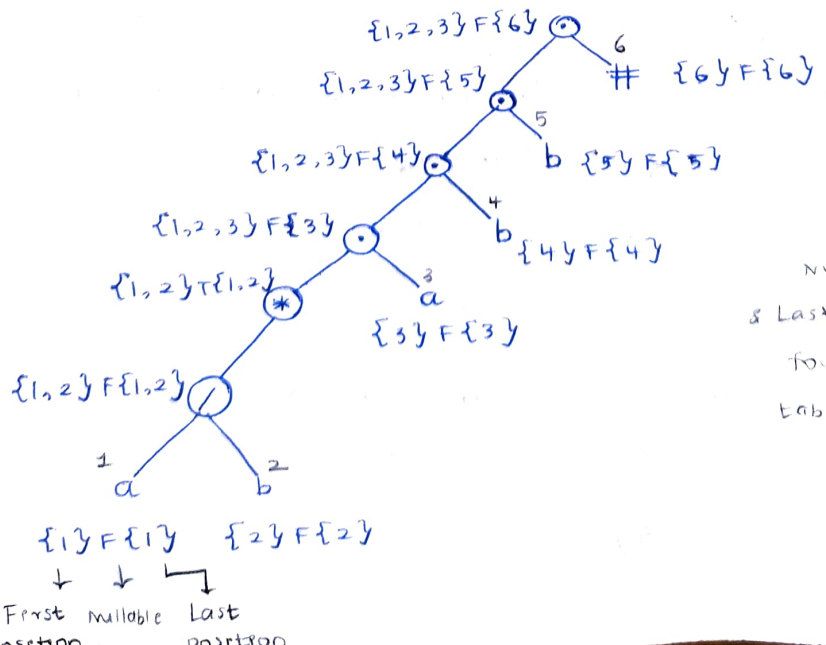
(ii) Syntax tree

Here operators are nodes and variables are leaf.



(After constructing syntax tree, all leaves node except ε should be numbered from 1 to n).

(iii)



Nullable, First position & Last position are found using the table given.

(iv) Finding follow positions for * and · node

Nodes	position
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	–

DFA transition

Initial state – {1, 2, 3} – A (Intha values ta ch tree ox compare panunum, athu tr la enga a vanthunu paakanum for Morc(A,

Move (A, a) = {1, 3}

Dtran (Move (A, a)) = Followpos(1) ∪ Followpos(3)

= {1, 2, 3} ∪ {4}

= {1, 2, 3, 4} → B

Move (A, b) = {2}

Dtran (Move (A, b)) = Followpos(2)

= {1, 2, 3} → A

Move (B, a) = {1, 3}

Dtran (Move (B, a)) = Followpos(1) ∪ Followpos(3)

= {1, 2, 3, 4} → B

Move (B, b) = {2, 4}

Dtran (Move (B, b)) = Followpos(2) ∪ Followpos(4)

= {1, 2, 3} ∪ {5}

= {1, 2, 3, 5} → C

Move (C, a) = {1, 3}

Dtran (Move (C, a)) = Followpos(1) ∪ Followpos(3)

= {1, 2, 3, 4} → B

Move (C, b) = {2, 5}

Dtran (Move (C, b)) = Followpos(2) ∪ Followpos(5)

= {1, 2, 3, 6} → D

Move (D, a) = {1, 3}

Dtran (Move (D, a)) = {1, 2, 3, 4} → B

Move (D, b) = {2}

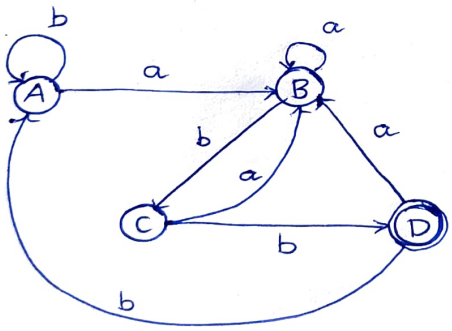
Dtran (Move (D, b)) = {1, 2, 3} → A

Transition table:

States	Input symbol	
	a	b
A	B	A
B	B	C
C	B	D
D	B	A

syntax tree la ethuvu # irukuvu paakanum, athu entha state lelaam iruko athu thaan final state

Transition Diagram:



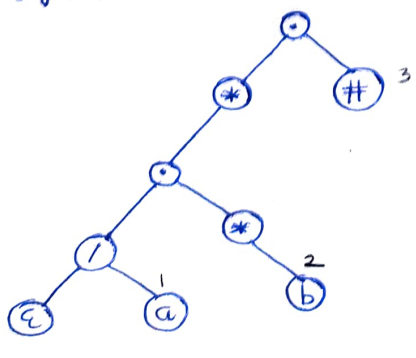
2) $((\varepsilon/a).b^*)^*$

Soln:

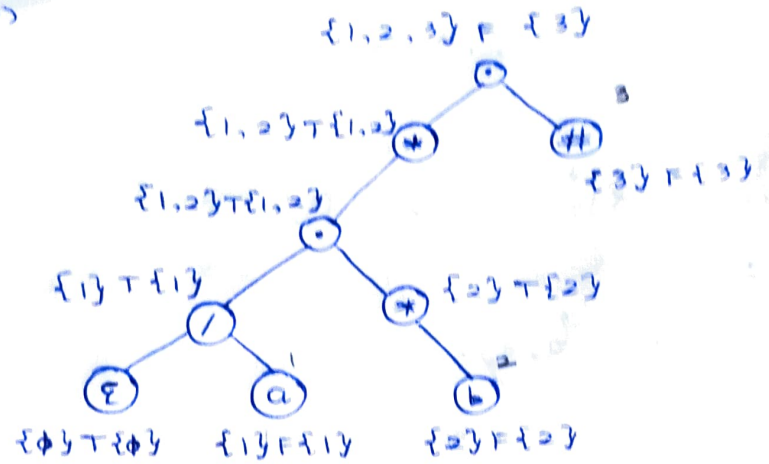
(i) Augmented grammar

$((\varepsilon/a).b^*)^*.\#$

(ii) Syntax tree



(iii)



Follow na last portion ku apply cha remaining portion.

(iv)

Nodes	position
1	{2, 1, 3}
2	{2, 1, 3}
3	—

Transition table

states	input symbol	
	a	b
A	A	A

Transition Diagram:

3) $a^*/b^*.a/b$ (* > . > /)

Soln: (i) Augumented Grammar

$a^*/b^*.a/b.\#$

(ii)