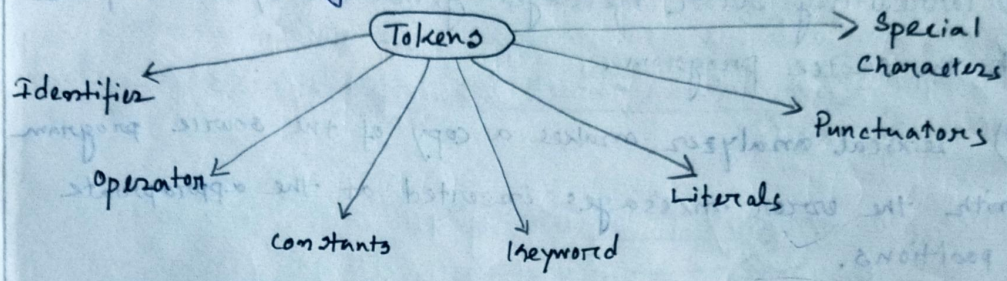
Introduction to Lexical Analyzer

Module II

At the first phase of compiler, the main task of lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is also called as Scanner.

- Scan the Pure HLL code line by line.
- Takes Lexemes as i/p and produces Tokens, uses DFA for pattern matching.



When the lexical analyzer discovers the type of lexemes, it needs to enter the lexeme into the symbol table. In some cases, information regarding the kind to identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

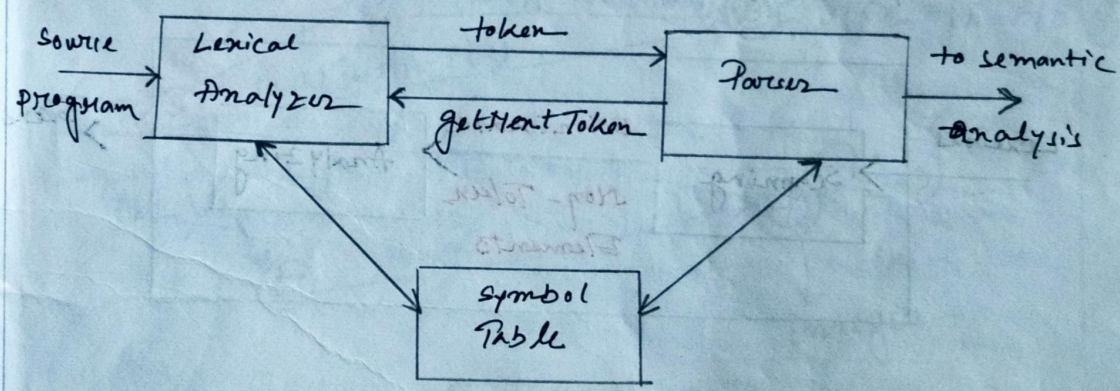


Fig 1: Interactions b/w the lexical analyzer and the parser

These interactions are suggested by Fig 1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the getNextToken command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

Basic functions

1) Reads the input program character by character, produces a stream of tokens which is used by the parser.

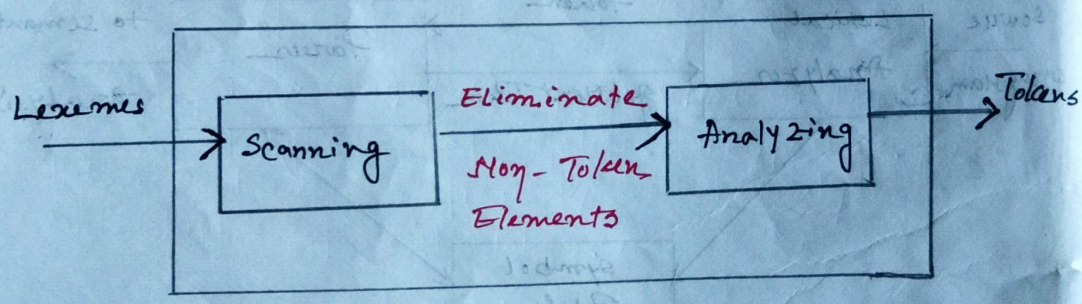
2) Stripping out the comments and whitespace (blank, newline, tab, and perhaps other characters that are used to seperate tokens in the input).

3) Correlating error messages generated by the compiler with the source program.

4) Lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions.

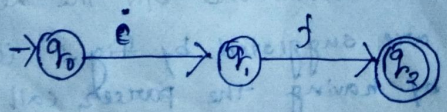
5) If the source program uses a macro-preprocessor, the expansion of the macros may be performed by the lexical analyzer.

Lexical Analyzer are divided into a cascade of two processes:

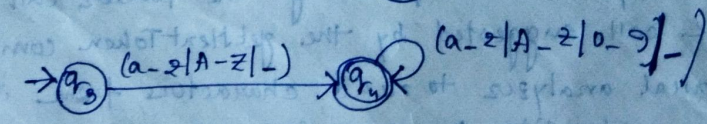


e- Tokens

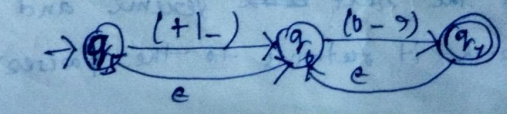
- if



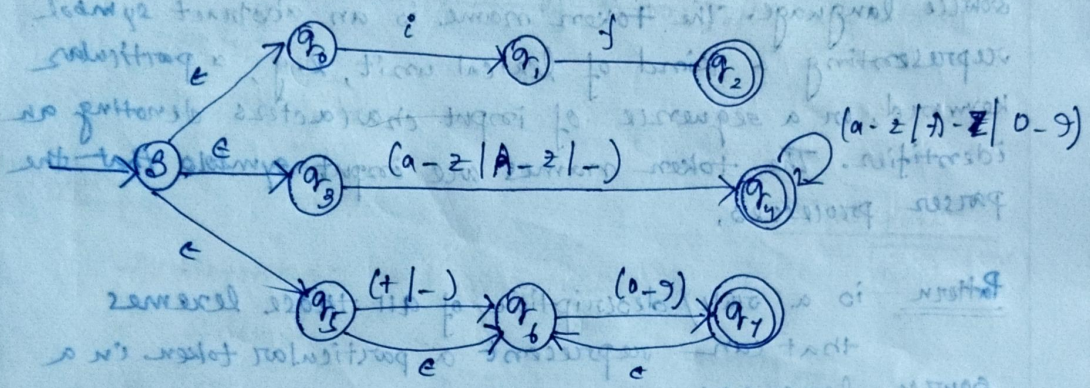
- identifier



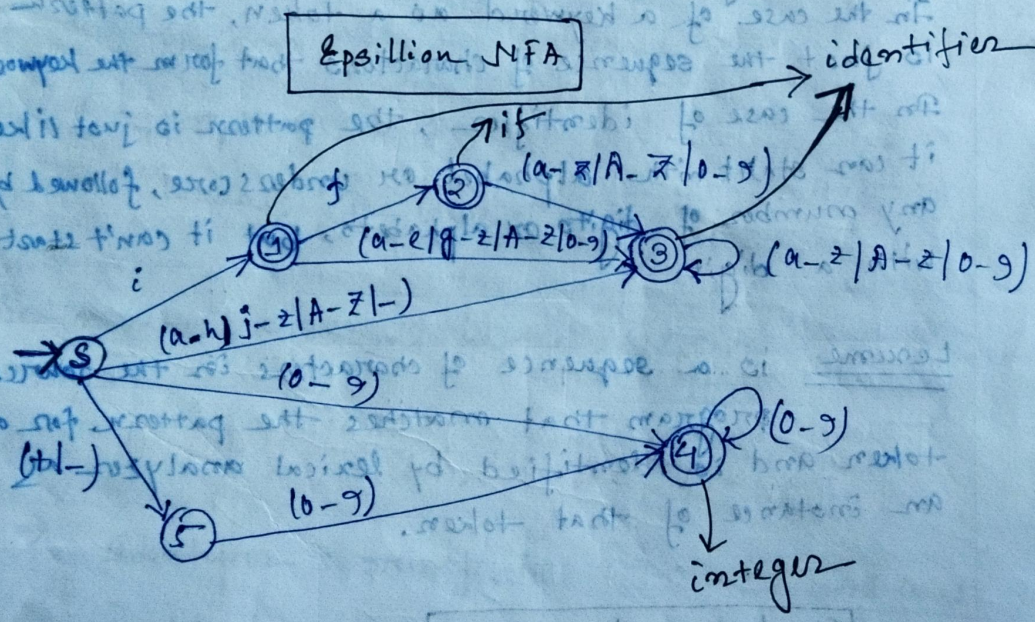
- integer



For the analyzing phase of lexical analyzer, we cann't have ~~a single~~ different machines for different types token. So, we need a single machine for that, combaining all of these.



Epsillion NFA



DFA

Tokens, Patterns and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

Tokens is a pair consisting of token name and optional attribute value. In simple terms Tokens are terminal symbols of the source language. The token name is an abstract symbol representing a kind of lexical unit, e.g. a particular keyword, or a sequence of input characters denoting an identifier. The token names are input symbols that the parser processes.

Pattern is a rule/description of all those lexemes that can represent a particular token in a source language.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. In the case of identifier, the pattern is just like it can start with alphabet or underscore, followed by any number of digits or alphabets, but it can't start with a digit.

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

$\boxed{\text{count = count + temp ;}}$

Tokens – identifier, operator, punctuations

Lexemes – count, temp, =, +, ;

Patterns are the rules to identifying the lexemes as particular tokens.

Lexical ~~Error~~ Analyzer – Tokenization

```
int main ()
{
    int x, a=2, b=3, c=5;
    x = a + b * c;
    printf ("The value of x is %d", x);
    return 0;
}
```

Tokens

Total count

39

1. Keyword: int (2), return
2. Identifier: main (2), x (2), a (2), b (2), c (2), printf
3. Punctuator: ((2),) (2), { (1), , (1), ; (4), } (1)
4. Operator: = (1), + (1), * (1)
5. Constant: 2, 3, 5, 0
6. Literal: "The value of x is %d"

Q. The number of tokens in the following C statement is:

```
printf ("i = %d, &i = %x", i, &i);
```

⇒ Identifier : printf (1), i (2)

Punctuator : ((2),), ; (2)

Total count

10

Literal : "i = %d, &i = %x"

Special character : &

Lexical Errors & Recovery

Error Handler in Lexical analysis phase mainly does three things → Error Detection, Error Reporting, and finally error recovery.

- Errors
  - Compile-time (Before the execution of the program)
    - Lexical Errors
    - Syntax Errors
    - Semantic Errors
  - Runtime (During the execution of the program)

→ Indentifiers that are way too long.

→ Finite State machine of the identifier that we observed permits identifiers to have as long ~~as~~ name as possible. But that is from logical ~~persp~~ perspective, nonetheless implementation wise there is a restriction on this. Because of the six phases present in the compiler, if only pattern matching in the lexical analysis takes ~~it~~ indefinite amount of time ~~will~~ that ~~the~~ will not effecient. So, during practical implemention of every language, different programming language set different rule.

C : 31 (ANSI)

247 (Microsoft C compiler)

C ++ : 2048 (Intel and microsoft)

Python : 79

→ Exceeding length of numeric constants.

int i = 4567891

size 2 bytes → variable range must be in b/w

-32768 to 32767

→ Numeric constants which are ill-formed.

int i = 4567$91;

error

→ Illegal characters that are absent from the source code.

char x[] = "compiler"; $

Recovery

• Panic-mode recovery → once an error has been encountered successive characters are skipped until valid delimiter is found.

→ int ANE$0;

→ while (condition)
{
}
skipped

• Replace a character with another.

itt NE$0; → int NE$0;

• Transpose of two adjacent characters.

union test
{
int x;
float y;
} T1;

→

union test
{
int x;
~~int~~
float y;
} T1;

• Insert a missing character.

it NE$0; → int NE$0;

• Delete an unknown or extra character.

intt NE$0; → int NE$0;

Input Buffering

→ Reading the source program for lexical analysis is a difficult task.

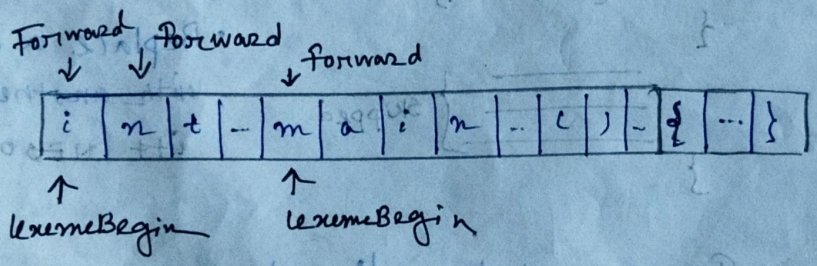
→ There will be situation where one need to look one or many characters beyond lexemes to recognize the end of it.

→ Eg: In C, single ~~characters~~ characters operators like, -, =, or < can be beginning of two character ~~programm~~ operators like, ->, ==, or <=.

→ To read the ~~s~~ lexemes character by character, lexical analyzer ~~use~~ uses two pointers.

- lexeme Begin Pointer - Points to the beginning of the current lexeme.
- Forward Pointer - Forward pointer reads the first character and move forward / right.

```
int main()
{
}
```



When the forward pointer ~~read~~ encounters a blank space, it identifies the end of the lexeme, ~~it~~ and then the lexeme is identified as token.

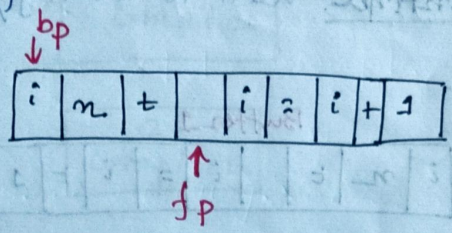
Then the lexical analyzer ignores the blank space and places ~~the~~ both of the pointer at ~~to~~ the ~~the~~ next lexeme.

→ Input character is read from the secondary storage at first, which is costly in terms of timing and efficiency. That is where buffering is used. A block of data is first read into a buffer, and then second by lexical analyzer.

→ **Single Buffer Input Scheme**

In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very ~~among~~ long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwritting the first of lexeme.

```
int i=i+1;
    j=j+1;
```

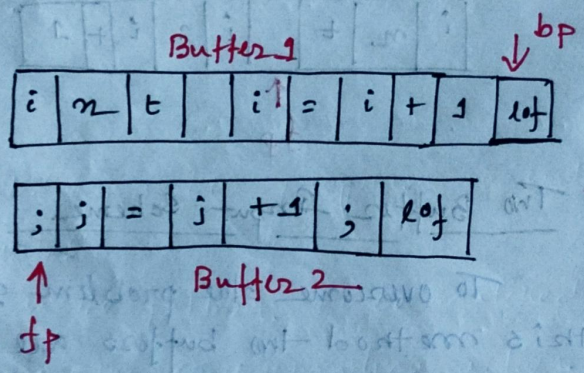


→ **Two Buffer Input Scheme**

To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. Two buffers are scanned alternately. When the end of current buffer is reached, the ~~other~~ other buffer is filled. The only problem ~~wh~~ with this method is that if length of lexeme is longer than length of the buffer then scanning input cannot be scanned completely.

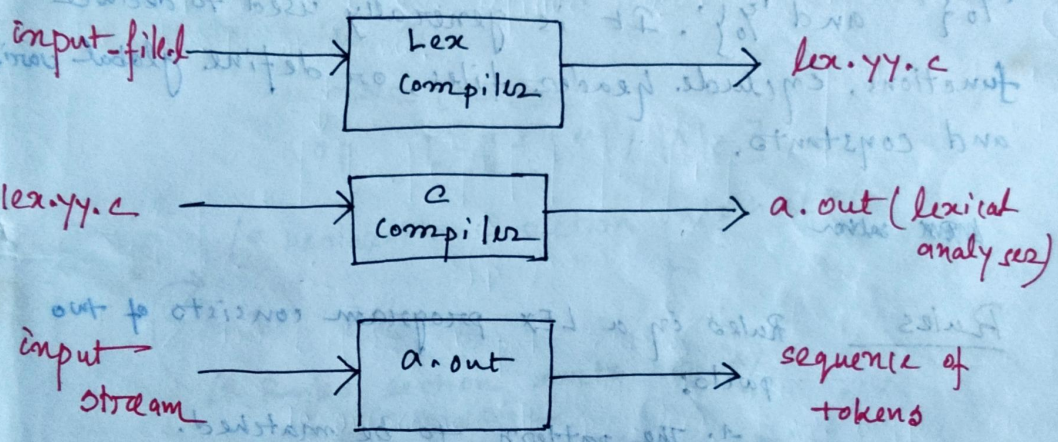
Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right to search end of lexeme. As soon as blank character is recognized, the string between bp and fp is identified as corresponding token. To identify, the boundary of first buffer end of buffer character (eof) should be placed at the end first buffer.

Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. When fp encounters first eof, then one can recognize end of the first buffer and hence filling up second buffer started. In the same way when second eof is obtained then it indicates the end of second buffer. Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified. The eof character introduced at the end is called as **Sentinel**, used to identify the end of buffer.



Design of a Lexical Analyzer – Lex Tool

Lex is a tool / program that generates lexical analyzer. It is used with YACC ~~par~~ parser generator. Technically, LEX translates a set of regular expression specifications (given as input in the input_file.l) into a a implementation of a corresponding finite state machine (lex.yy.c). This c program, when compiled, yields an executable lexical analyzer, (a.out). a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex was developed by Mike Lesk and Eric Schmidt at Bell Labs.

The Structure of Lex Programs.

A LEX program consists of three sections: Declaration, Rules and Auxiliary functions.

DECLARATIONS
%%
RULES
%%
AUXILIARY FUNCTIONS

2.1 Declarations

The declarations section consists of two parts, auxiliary declarations and regular definitions.

The auxiliary declarations are copied as such by LEX to the output lex.yy.c file. This C code consists of instructions to the C compiler and are not processed by LEX tool. The auxiliary declarations (which are optional) are written in C language and are enclosed within '%{' and '%}'. It is generally used to declare functions, include header files, or define global variables and constants.

Rules Rules in a LEX program consists of two parts:

1. The pattern to be matched.
2. The corresponding action to be executed.

Auxiliary functions

LEX generates C code for the rules specified in the rules section and places this code into a single function called Yylex().

Example

```
/* Declarations section starts here */
/* Auxiliary declarations start here */
%{
    #include <stdio.h>
    int global_variable;
%}
/* Auxiliary declarations end & Regular definitions
start here */
    number [0-9]+
    op     [-|+|*|/|^|=]
/* Declarations section ends here */
%%
/* Rules section starts here */
    regular expression
    {number}  {printf("number");}
    {op}      {printf("operator");}
%%
/* Rules section ends here */
/* Auxiliary section starts here */
%%
    int main()
    {
        yylex();
        return 1;
    }
%%
/* Auxiliary section ends here */
```

Regular Expression

Regular Expression is a way of representing Regular Language.

1. $L = \{0, 1, 2\}$

   → 0 or 1 or 2

   $R(L) = 0 + 1 + 2$

2. $L = \{\Lambda, ab\}$

   $R(L) = \Lambda ab$

3. $L = \{abb, a, b, bba\}$

   → abb or a or b or bba

   $R(L) = abb + a + b + bba$

4. $L = \{\Lambda, 0, 00, 000, \ldots\}$

   $R(L) \rightarrow \Lambda 0^*$

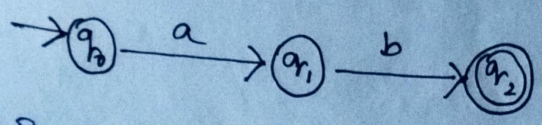
   $= 0^*$ (closure of 0)

5. $L = \{1, 11, 111, \ldots\}$

   $R(L) \rightarrow 1^+$ (kleene Plus)

Conversion of Finite Automata to Regular Expression.

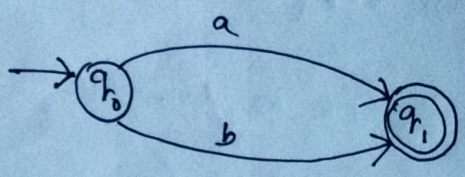
1.)



$R = ab$

Finite Automata is in Series form.

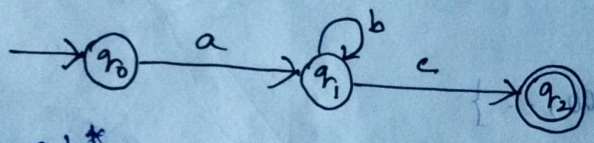
2)



$R = a+b$

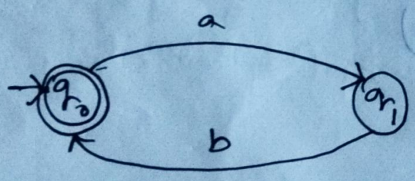
finite automata is in union form.

3.



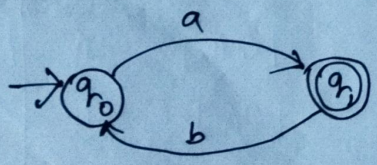
$R = ab^*c$ Loop is optional.

4.



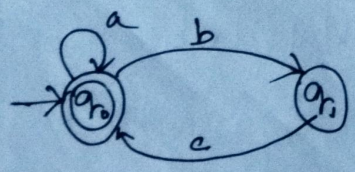
$R = (a.b)^*$

5.



$R = a(ba)^*$

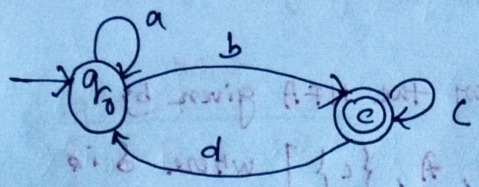
6)



$R = (a^* + (bc)^*)^*$

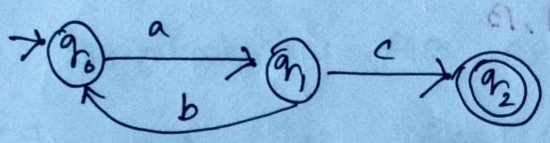
$= (a+bc)^*$

8)



$R = a^* b (c + da^* b)^*$

9)



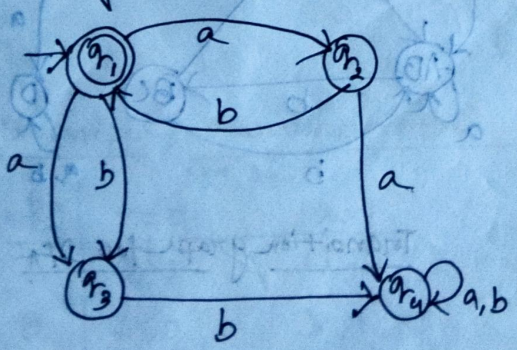
$R = a(ba)^* c$

10)

$R = a(b + (a)^* d$

~~Conversion of RE to DFA~~

using Arden's Theorem



$q_1 = \epsilon + q_2 b + q_3 a$ — (1)

$q_2 = q_1 a$ — (2)

$q_3 = q_1 b$ — (3)

$q_4 = q_2 a + q_3 b + q_4 a + q_4 b$ — (4)

$q_1 = \epsilon + q_2 b + q_3 a$

$= \epsilon + q_1 ab + q_1 ba$

$= \epsilon + q_1 (ab + ba)$

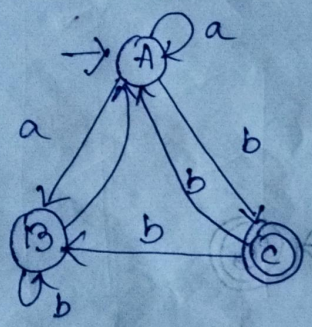
$= \epsilon (ab + ba)^* = (ab + ba)^*$

Conversion of NFA to DFA

Find the equivalent DFA for the NFA given by $M = [\{A, B, C\}, (a, b), \delta, A, \{C\}]$ where δ is given by

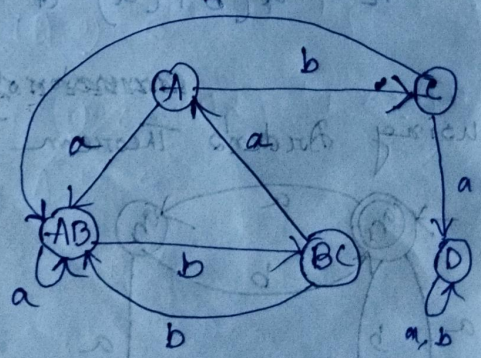
	a	b
→ A	A, B	C
B	A	B
C	–	A, B

Transition graph for NFA



Transition Table for DFA

	a	b
→ A	[AB]	C
[AB]	[AB]	[BC]
[BC]	A	[AB]
C	D	[AB]
D	D	D



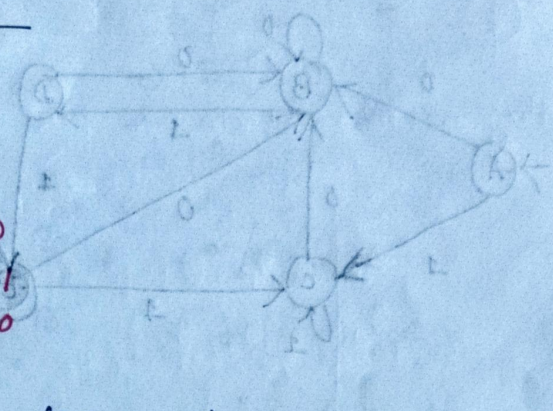
Transition graph for DFA

* D → is the dead state.

Minimize ~~the~~ FSM

Minimize the following given machine using State Equivalence method.

PS	NS, Z x=0	x=1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0



$P_0 \rightarrow 0$ Equivalent class (ABCDEF)

$P_1 \rightarrow 1$ " " (ACE) (BDF)

$P_2 \rightarrow 2$ " " (AC) (BD) (F)

$P_3 \rightarrow 3$ " " (AC) (E) (BD) (F)

$P_4 \rightarrow 4$ " " (AC) (E) (BD) (F)

Let,

$AC = \alpha$

$E = \beta$

$BD = \gamma$

$F = \delta$

Intermidate Table

PS	NS, Z x=0	x=1
$AC \rightarrow \alpha$	β, 0	γ, 1
$E \rightarrow \beta$	α, 0	δ, 1
$BD \rightarrow \gamma$	δ, 0	γ, 0
$F \rightarrow \delta$	γ, 0	α, 0

Canonical / Standard form

PS	NS, Z x=0	x=1
A	B, 0	C, 1
B	A, 0	D, 1
C	D, 0	A, 0
D	C, 0	A, 0

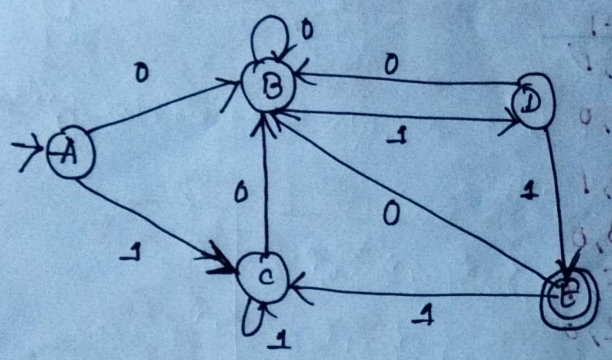
$\alpha = A$

$\beta = B$

$\gamma = C$

$\delta = D$

Minimize the following DFA using Myhill Nerode Theorem.



	A	B	C	D	E
A					
B	✓				
C		✓			
D	✓	✓	✓		
E	✓	✓	✓	✓	

1st Round

(B, A) = δ(B, 1) = D δ(B, 0) = B
δ(A, 1) = C δ(A, 0) = B

(C, A) = δ(C, 0) = B δ(C, 1) = C
δ(A, 0) = B δ(A, 1) = C

(C, B) = δ(C, 0) = B δ(C, 1) = C
δ(B, 0) = B δ(B, 1) = D

(D, A) = δ(D, 0) = B δ(D, 1) = E
δ(A, 0) = B δ(A, 1) = C

(D, B) = δ(D, 0) = B δ(D, 1) = E
δ(B, 0) = B δ(B, 1) = B

$(D, E) = \delta(D, 0) = B \qquad \delta(D, 1) = E$

$\delta(E, 0) = B \qquad \delta(E, 1) = C$

2nd Round

$(B, A) = \delta(B, 0) = B \qquad \delta(B, 1) = D$

$\delta(A, 0) = B \qquad \delta(A, 1) = C$

$(C, A) = \delta(C, 0) = B \qquad \delta(C, 1) = C$

$\delta(A, 0) = B \qquad \delta(A, 1) = C$

$(C, B) = \delta(C, 0) = B \qquad \delta(C, 1) = C$

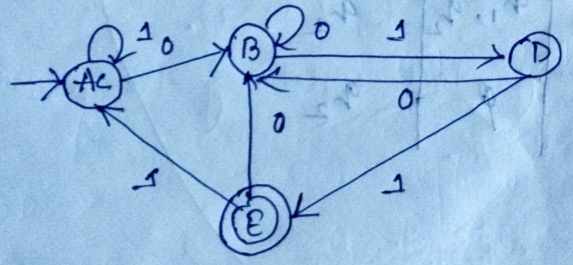
$\delta(B, 0) = B \qquad \delta(B, 1) = D$

3rd Round

$(C, A) = \delta(C, 0) = B \qquad \delta(C, 1) = C$

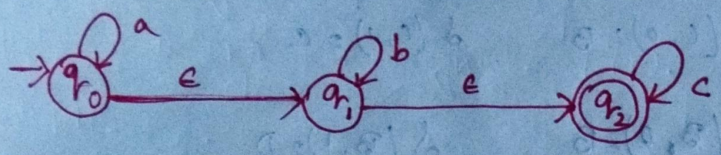
$\delta(A, 0) = B \qquad \delta(A, 1) = C$

Minimized Pairs are: (A, C), B, D, E



Minimized DFA

Conversion of ε-NFA to NFA



Finding out the null closure

Q	ε - closure
q_0	$\{q_0, q_1, q_2\}$
q_1	$\{q_1, q_2\}$
q_2	$\{q_2\}$

Formula for conversion

$$\delta'(q_i, x) = \epsilon\text{-closure}[\delta[\epsilon\text{-closure}(q_i), x]]$$

States of NFA	a	b	c
q_0	q_0, q_1, q_2	q_1, q_2	q_2
q_1	ϕ	q_1, q_2	q_2
q_2	ϕ	ϕ	q_2

$$\delta'(q_0, a) = \epsilon\text{-closure}[\delta[q_0, q_1, q_2], a]$$

$$= \epsilon\text{-closure}[q_0]$$

$$= [q_0, q_1, q_2]$$

$$\delta'(q_0, b) = \epsilon\text{-closure}[\delta(q_0, q_1, q_2), b]$$

$$= \epsilon\text{-closure}[q_1]$$

$$= [q_1, q_2]$$

$$\delta'(q_0, c) = \epsilon\text{-closure}[\delta(q_0, q_1, q_2), c] = [q_2]$$

$$\delta'(q_1, a) = \epsilon\text{-closure}[\delta(q_1, q_2), a]$$

$$= \epsilon\text{-closure}[\phi]$$

$$= \phi$$

$$\delta'(q_1, b) = [q_1, q_2]$$

$$\delta'(q_1, c) = [q_2]$$

$$\delta'(q_2, a) = \phi$$

$$\delta'(q_2, b) = \phi$$

$$\delta(q_2, c) = q_2$$

Regular Expression to DFA (Directly)

Input :- A regular expression r.

Output :- A DFA that recognizes $L(r)$.

Method :-

1. Construct a syntax tree T from augmented regular expression $(r)\#$.
2. Compute nullable, firstpos, lastpos, and followpos for T.
3. Construct Dstates (the set of states of DFA D) and Dtran (transition function for D).

Note :-

- Dstates are sets of positions in T.
- Initially, each state is unmarked.
- state becomes "marked" just before we consider its out-transitions.
- start is firstpos (root in T).
- Accepting start containing the position for endmark '#'.

Important definition / terminologies

Important states of NFA

A state of an NFA is important if it has a non-ϵ out transition.

In subset construction algorithm, only the important states in a set T are used when it computes ϵ-closure (move $[T, a]$) i.e, the set of states reachable from T on input a.

Each important state correspond to a particular operand in the RE.

• The constructed NFA has only 1 accepting state, but it is not important as, it's having no out-transition.

Augmented Regular Expression

Concatenated a unique right endmarker # to r, we give a transition on # for accepting state.

So, NFA is now having important states.

Syntax tree

Regular expression is represented as a syntax tree (post-fix).

Leaves corresponds to operands.

Interior nodes → operators.

Interior nodes called →

i) cat-node ← (concatenation (·) operator)

ii) or-node ← (union (|))

iii) star-node ← (star (*))

- cat-nodes represented by hollow circles.
- Leaves are represented by either ε or alphabet.
- Each leaf is numbered by integer (except ε).
- These position values correspond to important states of NFA.

Functions computed from the syntax tree

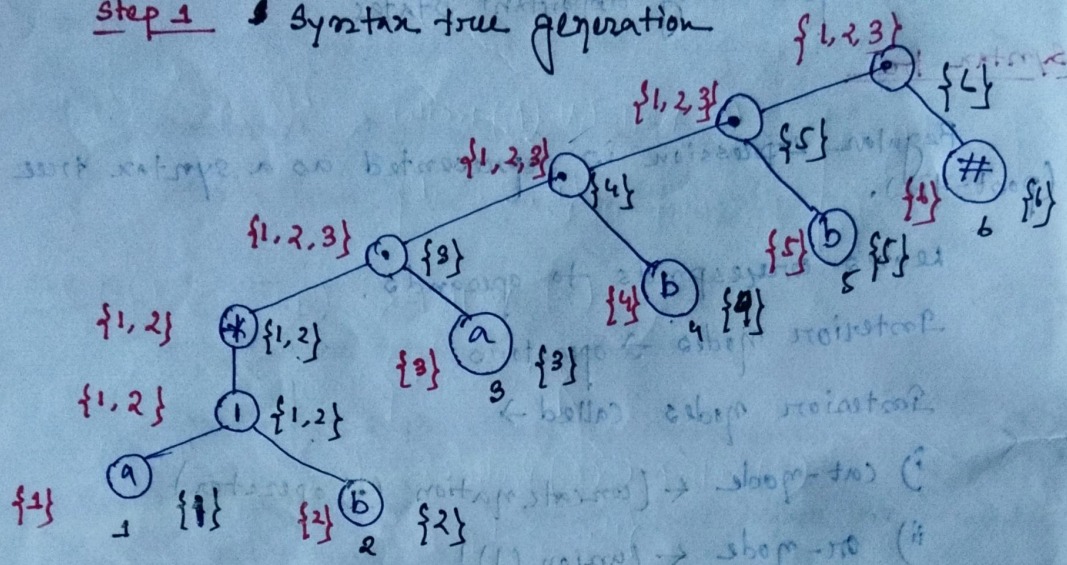
1. Nullable (n) → iff the subexpression represented by n has ε in ~~into~~ its language, i.e the subexpression can be made null or empty.

2. First-pos (n) → set of positions in the sub-tree rooted n that gives the set of first positions that can match first symbol of a string generated by the sub expression.

3. **lastpos(n)** → is the set of positions in the subtree rooted at n that correspond to the last symbol of at least one string in the language of the subexpression rooted at n.

Example $(a|b)^* a.b.b\#$ (Augmented RE)

Step 1 Syntax tree generation



$(a|b)^* . a . b . b \#$
4 2 3 4 5 6

here ⊛ node is nullable

First pos
Last pos

Followpos Table

Only for ⊛ and cat-node (•)

For ⊛ node

for (each i in lp(n))
followpos(i) = followpos(i) ∪ fp(n)

For cat-node (•)

if (n == $c_1 . c_2$)
for (each i in lp(c_1))
followpos(i) = followpos(i) ∪ fp(c_2)

(i) $\circledast \rightarrow i$ in $fp(n) = \{1,2\}$

follow (1) $= \{\phi\} \cup fp(\circledast)$

$= \{\phi\} \cup \{1,2\}$

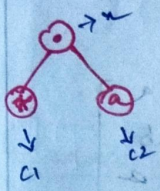
$= \{1,2\}$

follow (2) $= \{\phi\} \cup fp(n)$

$= \{\phi\} \cup \{1,2\}$

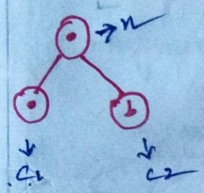
$= \{1,2\}$

(ii) i in $fp(c_1) = \{1,2\}$



follow (1) $= \{1,2\} \cup \{3\} = \{1,2,3\}$

follow (2) $= \{1,2\} \cup \{3\} = \{1,2,3\}$



i in $fp(c_1) = \{3\}$

follow (3) $= \{4\}$

$fp(c_1) = \{4\}$

follow (4) $= \{5\}$

$fp(c_1) = \{5\}$

follow (5) $= \{6\}$

	Node	FollowPos
a	1	{1, 2, 3}
b	2	{1, 2, 3}
a	3	{4}
b	4	{5}
b	5	{6}
#	6	{φ}

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

1 2 3 4 5 6

Now converting RE to DFA (directly)

Rule (i) initial state = fp (root)

(ii) Final state = node having position no. of #.

(iii) make new state with the help of followpos and node numbers for each symbol transition.

For ex:

$\rightarrow$ (1, 2, 3) initial state

Now, for generating next state,
first state combines the states (1, 2, 3)
So, for node (1, 2, 3) find the follow positions and make union individually for each symbol (a, b).

Follow(1) = {1, 2, 3}, Follow(2) = {1, 2, 3}, Follow(3) = {4}

for, i/p : "a"

δ({1, 2, 3}, a) → Follow(1) ∪ Follow(3)
= {1, 2, 3, 4}

pos 1 = a
pos 2 = b
pos 3 = a

δ({1, 2, 3}, b) → Follow(2)
= {1, 2, 3}

δ({1, 2, 3, 4}, a) → Follow(1) ∪ Follow(3)
= {1, 2, 3, 4}

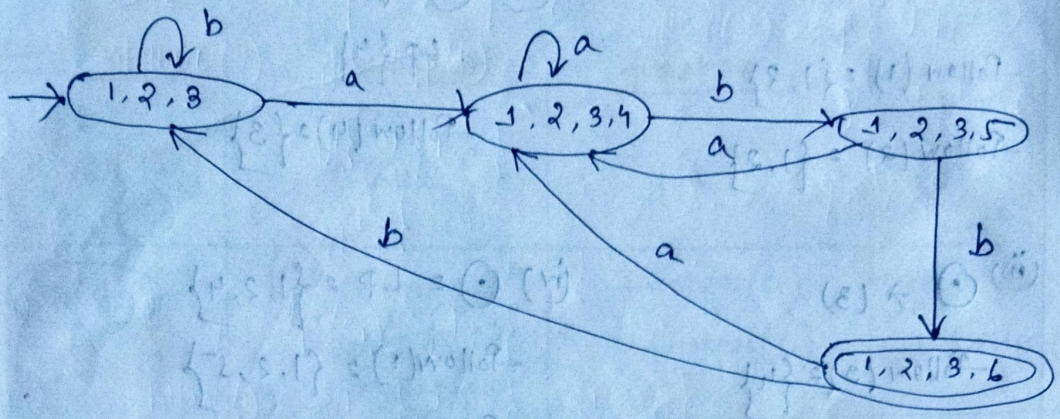
δ({1, 2, 3, 4}, b) → Follow(2) ∪ Follow(4)
= {1, 2, 3, 5}

δ({1, 2, 3, 5}, a) → Follow(1) ∪ Follow(3)
= {1, 2, 3, 4}

δ({1, 2, 3, 5}, b) → Follow(2) ∪ Follow(5)
= {1, 2, 3, 6}

δ({1, 2, 3, 6}, a) = {1, 2, 3, 4}

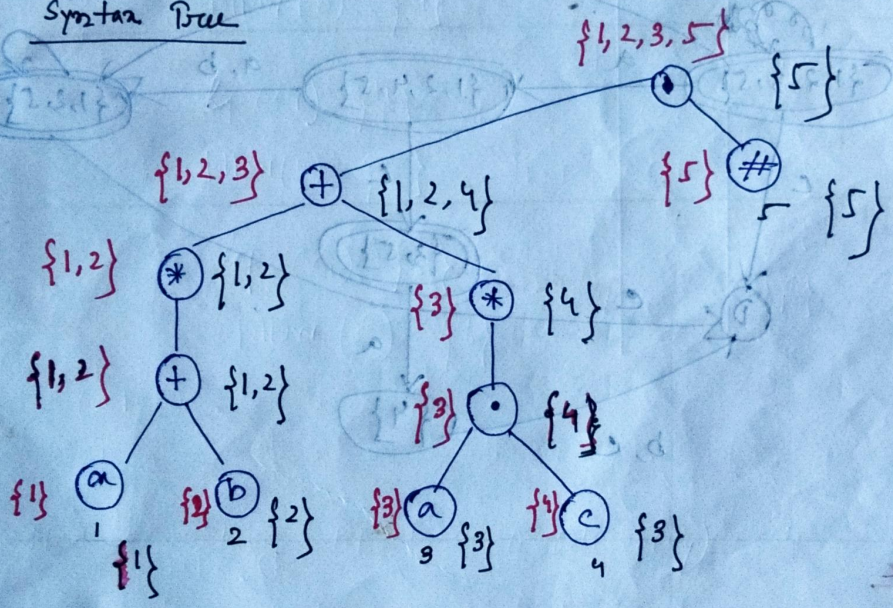
δ({1, 2, 3, 6}, b) = {1, 2, 3}



Example -2 $(a+b)^* + (a.c)^*$

Augmented RE – $(a+b)^* + (a.c)^* \#$

Syntax Tree



Followpos Table

	Node	Followpos
a	1	{1, 2, 5}
b	2	{1, 2, 5}
a	3	{4}
c	4	{3, 5}
#	5	∅

(i) ⊛ nodes

Follow (1) = {1, 2}

Follow (2) = {1, 2}

(ii) ⊙ → (3)

Follow (3) = {4}

(iii) ⊛ →

LP {4}

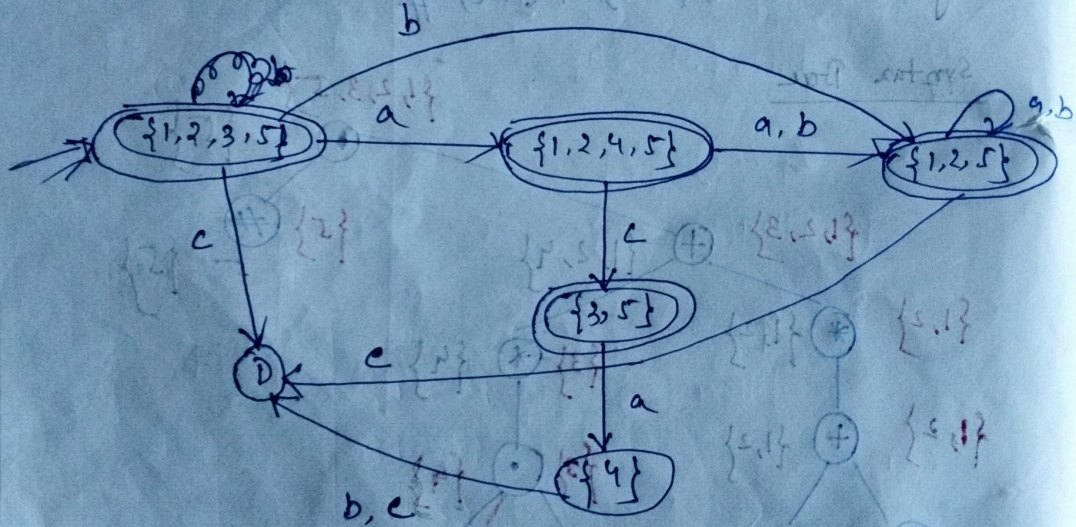
Follow (4) = {3}

(iv) ⊙ = LP = {1, 2, 4}

Follow(1) = {1, 2, 5}

Follow (2) = {1, 2, 5}

Follow (4) = {3, 5}



Exercise

Convert the Regular Expression to corresponding DFA (direct method)

1. $ab^*a(aba|ba)a^*$

2. $a^*b^*a(a|b)^*b^*a$

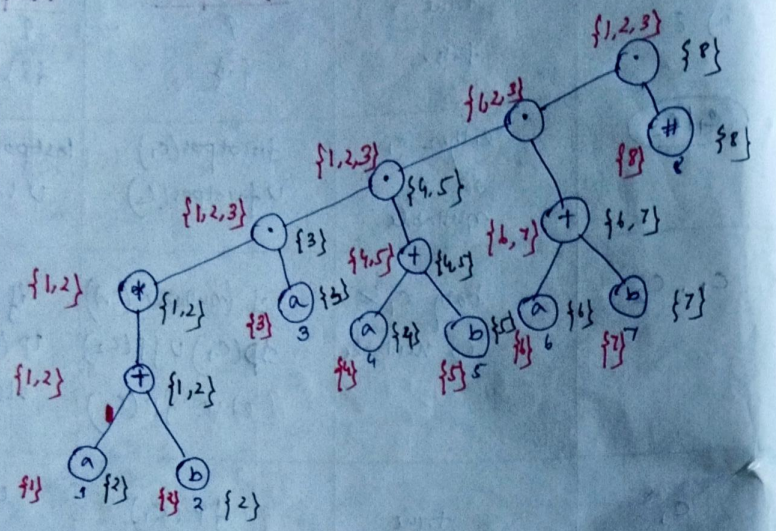
3. $(a|b)^*abb$

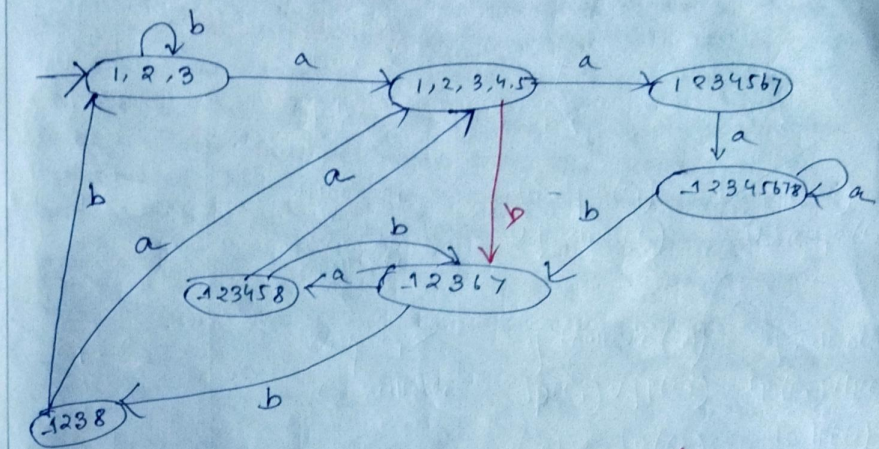
4. $(a+b)^*a(a+b)(a+b)\#$

Algorithm to calculate nullable, firstpos, lastpos.

Leaf node	nullable	firstpos	lastpos
i) ϵ	true	ϕ	ϕ
ii) i	false	$\{i\}$	$\{i\}$
$c_1 \mid c_2$	either c_1 OR c_2 nullable	firstpos(c_1) $\cup$ firstpos(c_2)	lastpos(c_1) $\cup$ lastpos(c_2)
$c_1 . c_2$	both c_1 & c_2 nullable	if (nullable(c_1)) fp(c_1) $\cup$ fp(c_2) else fp(c_1)	if nullable(c_2) lp(c_1) $\cup$ lp(c_2) else lp(c_2)
c_1^*	true	fp(c_1)	lp(c_1)

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



followpos Table

		Follow Set
a	1	{1,2,3}
b	2	{1,2,3}
a	3	{4,5}
a	4	{6,7}
b	5	{6,7}
a	6	8
b	7	8
#	8	ϕ