Lexical Analyzer

Phase 1: Lexical Analysis

- Lexical Analysis is the first phase when compiler scans the source code.
- This process can be left to right, character by character, and group these characters into tokens.
- The character stream from the source program is grouped into meaningful sequences by identifying the tokens.
- It makes the entry of the corresponding tokens into the symbol table and passes that token to next phase.

The primary functions of this phase are:

- Identify the lexical units in a source code
- Classify lexical units into classes like constants, reserved words, and enter them in different tables.
- Identify token which is not a part of the language.
- It will Ignore comments in the source program. It is accountable for terminating the comments and white spaces from the source program.

Tokens

- Tokens are the fundamental building blocks of a program's grammatical structure that represents such basic elements as identifiers, numeric literals, and specific keywords and operators of the language.
- It is basically a sequence of characters that are treated as a unit as it cannot be further broken down.
- In programming languages like C language- keywords (int, char, goto, continue, etc.) identifiers, operators (+, -, *, ….), delimiters/punctuators like comma (,), semicolon(;), etc. , strings can be considered as tokens.
- This phase recognizes following types of tokens:
    - Terminal Symbols (TRM)
    - Keywords and Operators,
    - Literals (LIT), and Identifiers (IDN).

Example

int a = 10; //Source code

Tokens

int (keyword),
a (identifier),
= (operator),
10 (constant)
; (punctuation-semicolon)

There are total 5 tokens in the above code line.

Lexeme

It is a sequence of characters in the source code that are matched by given predefined language rules for every lexeme to be specified as a valid token.

Example:
Lexeme:

main - identifier (token)
(,),{,} - punctuation (token)

Lexemes are the character strings assembled from the character stream of a program, and the token represents what component of the program's grammar they constitute.

Pattern

It specifies a set of rules that a scanner follows to create a token.

Example of Programming Language (C, C++):

For a **keyword** to be identified as a valid token, the pattern is the sequence of characters that make the keyword.

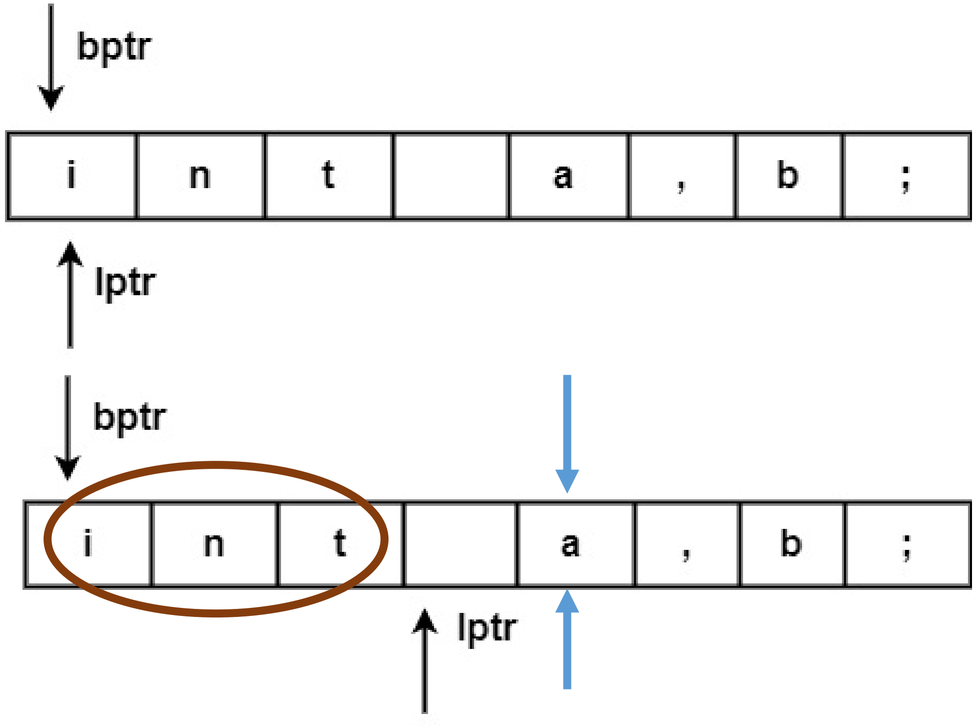
For **identifier** to be identified as a valid token, the pattern is the predefined rules that it must start with alphabet, followed by alphabet or a digit.

Input Buffering

Lexical Analysis scans input string from left to right one character at a time to identify tokens. It uses two pointers to scan tokens –

- Begin Pointer (bptr) – It points to the beginning of the string to be read.
- Look Ahead Pointer (lptr) – It moves ahead to search for the end of the token.

Example:



Recognition of Tokens

- Tokens can be recognized by Finite Automata.
- Generally, Finite Automata are required to implement regular expressions.
- It recognizes the various tokens with the help of regular expressions and pattern rules and classifies them.
- So, Tokens are recognized by regular grammar and are implemented by finite automata.

Examples

Input alphabet = {a,b}

Languages defined:

- (a|b) = {a,b}
- (a|b)(a|b) = {aa, ab, ba, bb}
- a* = {ε, a, aa, aaa, …..}
- (a|b)* = {ε, a, b, aa, ab, ba, bb, aaa…..}

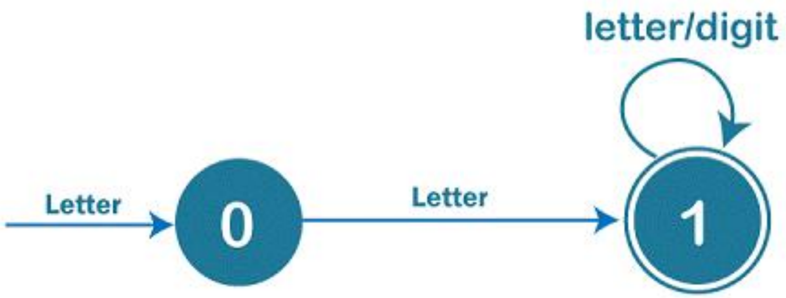
Regular Definitions

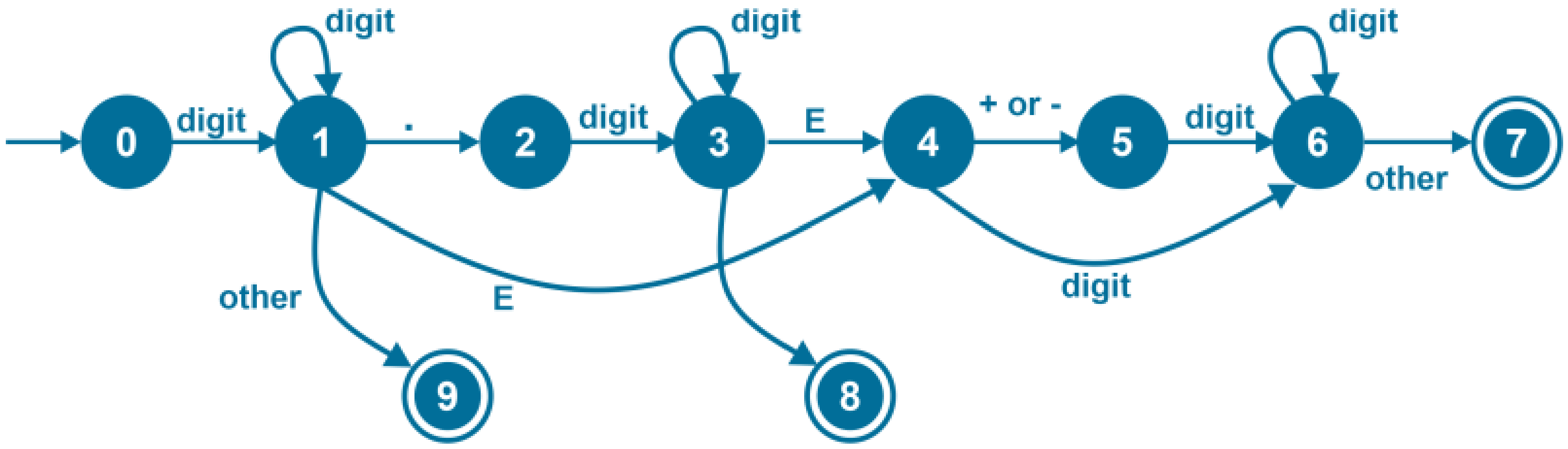
- identifier -> letter(letter|digit)* //For identifiers
- letter -> A|B|….|Z|a|b|….|z|_
- digit -> 0|1|…|9

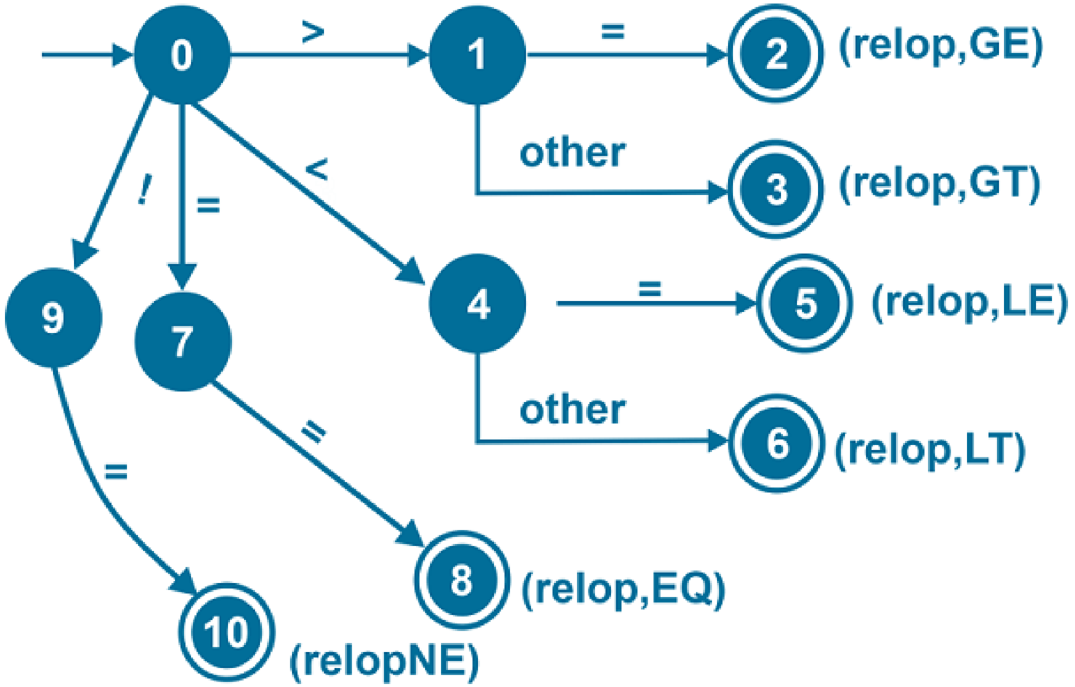
- digit -> 0|1|….9 //For Numbers
- digits -> digit (digit)*
- number -> digits (.digits)? (E[+ -] ? digits)?
- number -> digit+ (.digit)+? (E[+ -] ? digit+)?

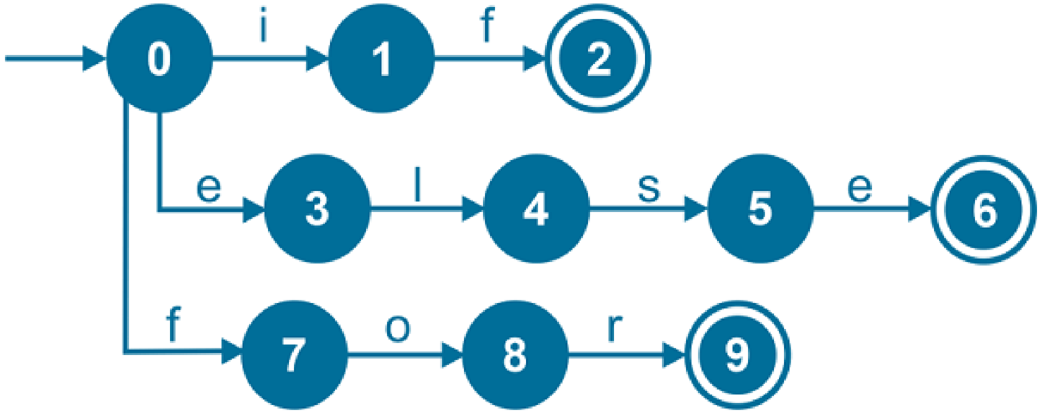
- delimiter -> ' ', '\t', '\n’ //For delimiters
- delimiters -> delimiter (delimiter)*

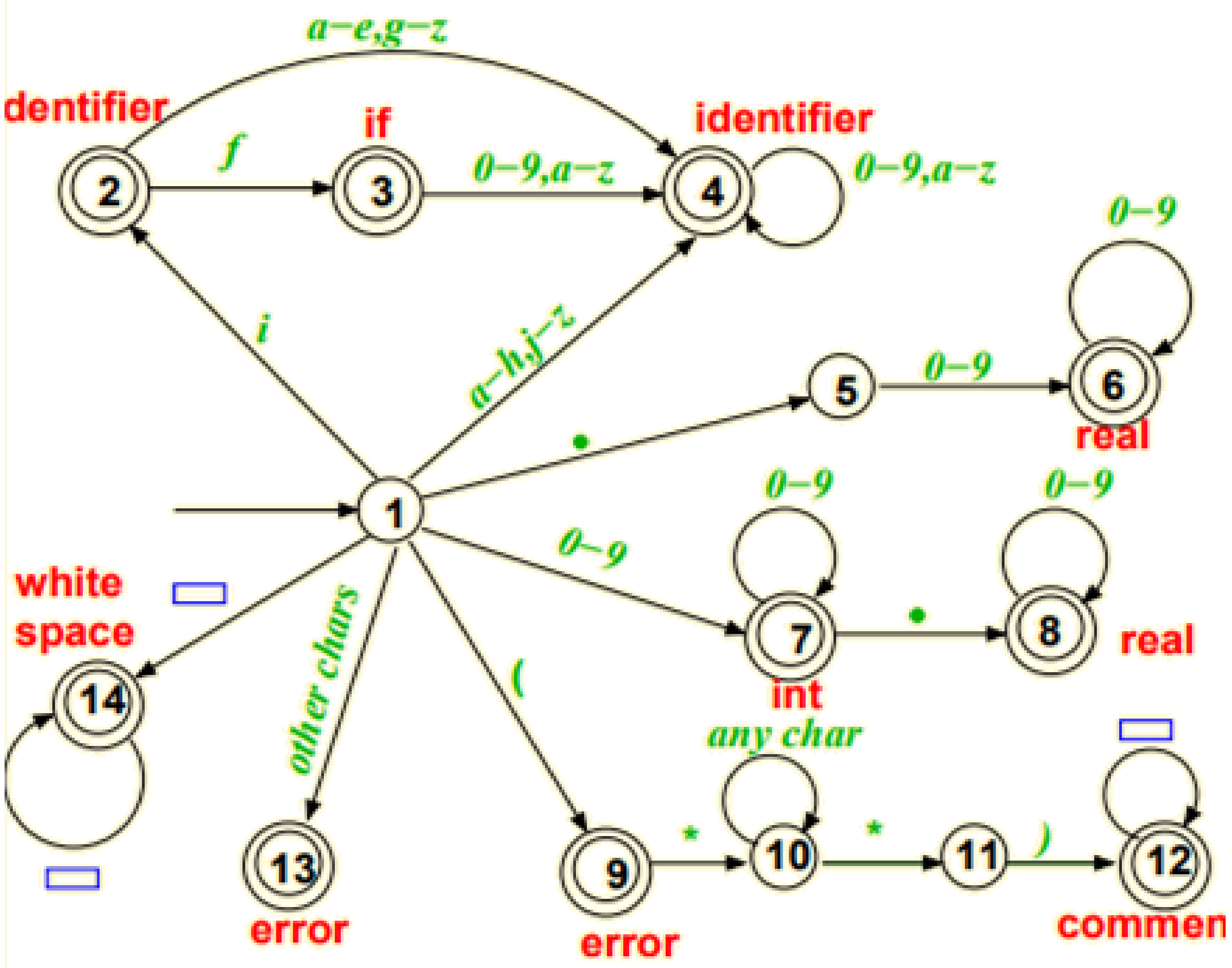
Transition Diagrams






0
>
1
=
2
(relop,GE)
other
3
(relop,GT)
<
4
=
5
(relop,LE)
other
6
(relop,LT)
!
9
=
7
=
8
(relop,EQ)
=
10
(relopNE)


0
i
1
f
2
e
3
l
4
s
5
e
6
f
7
o
8
r
9



a-e,g-z
dentifier
f
if
0-9,a-z
identifier
0-9,a-z
i
a-h,j-z
0-9
5
0-9
6
real
1
0-9
0-9
0-9
white
space
other chars
(
7
.
8
real
int
any char
14
13
error
9
error
*
10
*
11
)
12
commen
2
3
4

Finite Automata

The finite automata or finite state machine is an abstract machine that has five elements or tuples.

A Finite Automata consists of the following:

$\{ Q, \Sigma, q, F, \delta \}$

Q : Finite set of states.

Σ : set of Input Symbols.

q : Initial state.

F : set of Final States.

δ : Transition Function.

Based on the states and the set of rules the input string can be either accepted or rejected.

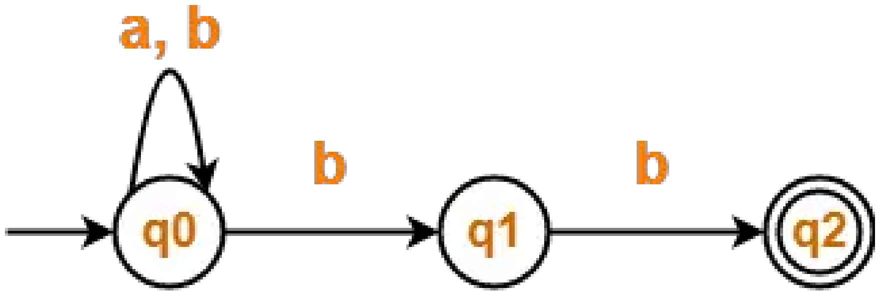
NFA and DFA

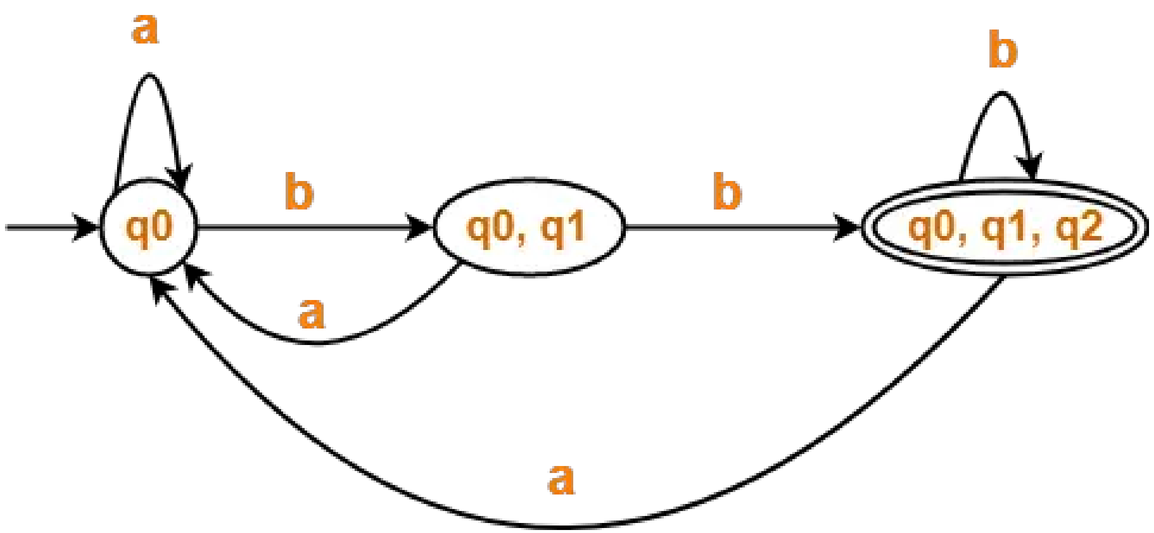
<u>Non-deterministic Finite Automata (NFA)</u>

- More than one transition occurs for any input symbol from a state.
- Transition can occur even on empty string (Ɛ).

<u>Deterministic Finite Automata (DFA)</u>

- For each state and for each input symbol, exactly one transition occurs from that state.





Conversion from RE to DFA

Regular Expression: r = (a|b)*abb

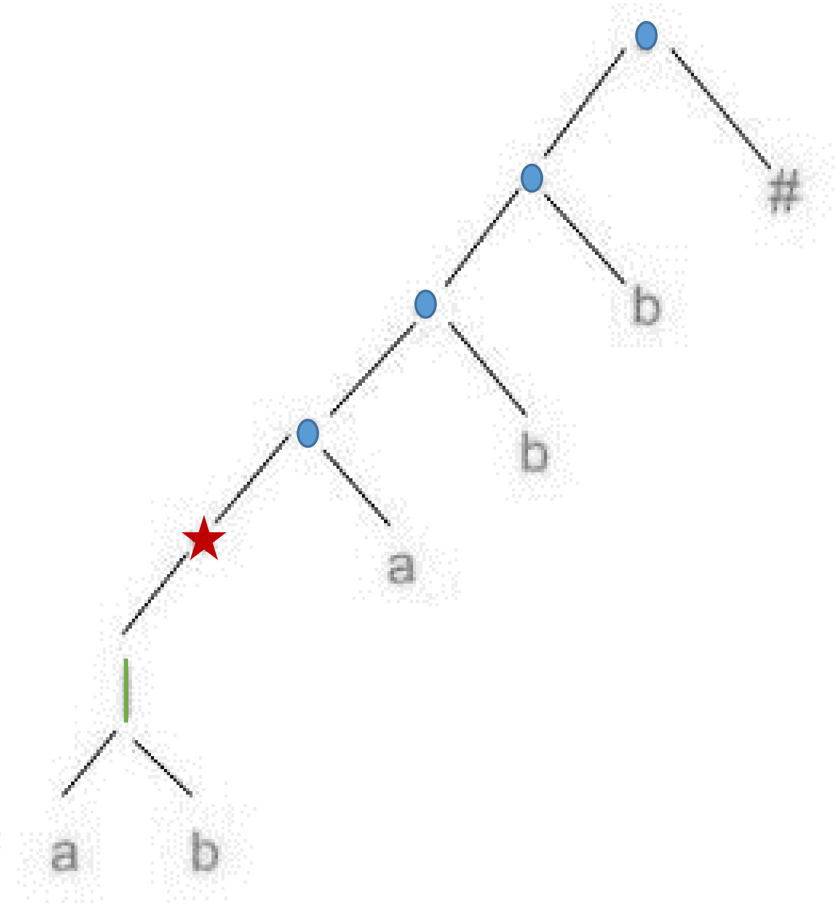
Step1: Construct augmented regular expression by adding right end marker (#) at the end of expression r.

r’ = (a|b)*abb#

Step2: Construct the syntax tree for r# where leaves correspond to operands and the interior nodes correspond to operators.

An interior node is called a cat-node if labeled by the concatenation operator(.), or-node if labeled by union operator(|), or star-node if labeled by star operator (*).

Syntax Tree for (a|b)*abb#



Step3: Compute nullable, firstpos, lastpos, and followpos

1. nullable(n) is true for a syntax tree node n if and only if the sub-expression represented by n has ε in its language.
2. firstpos(n) gives the set of positions that can match the first symbol of a string generated by the subexpression rooted at n.
3. lastpos(n) gives the set of positions that can match the last symbol of a string generated by the subexpression rooted at n.
4. followpos(p) for a position p, is the set of position q in the entire syntax tree such that there is some string $x = a_1a_2\ldots a_n$ in L((r)#) such that for some i, there is a way to explain the membership of x in L((r)#) by matching a_i to position p of the syntax tree and a_{i+1} to position q.

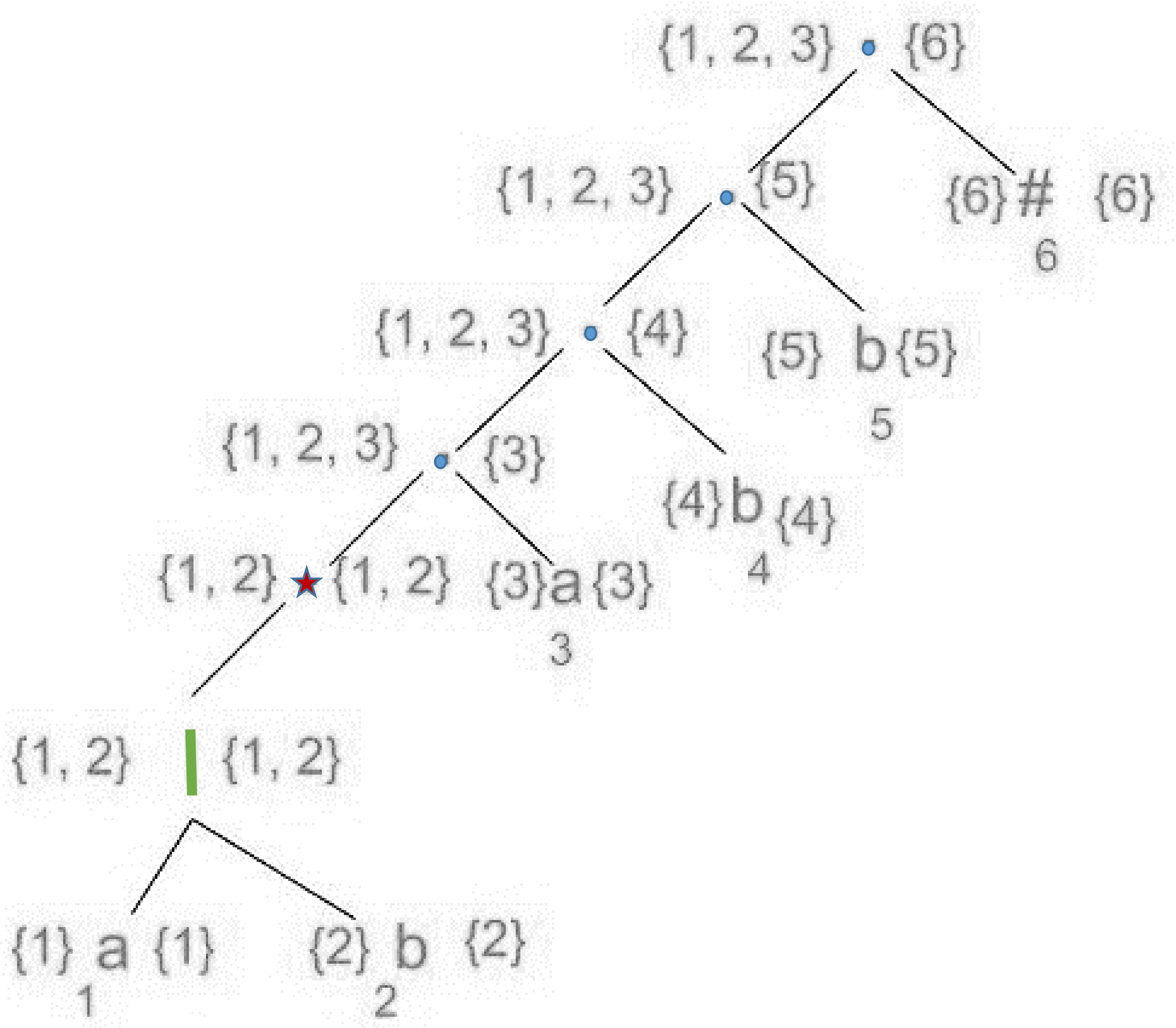
Computing followpos

1. If n is a cat-node with left child c1 and right child c2 and i is a position in lastpos(c1), then all positions in firstpos(c2) are in followpos(i).

2. If n is a star-node and i is a position in lastpos(n), then all positions in firstpos(n) are in followpos(i).

Node n	nullable(n)	firstpos(n)	lastpos(n)
n is a leaf node labeled ε	true	∅	∅
n is a leaf node labelled with position i	false	{ i }	{ i }
n is an or-node with left child c1 and right child c2	nullable(c1) or nullable(c2)	firstpos(c1) ∪ firstpos(c2)	lastpos(c1) ∪ lastpos(c2)
n is a cat-node with left child c1 and right child c2	nullable(c1) and nullable(c2)	If nullable(c1) then firstpos(c1) ∪ firstpos(c2) else firstpos(c1)	If nullable(c2) then lastpos(c2) ∪ lastpos(c1) else lastpos(c2)
n is a star-node with child node c1	true	firstpos(c1)	lastpos(c1)

firstpos and lastpos for nodes in the syntax tree for (a|b)*abb#



followpos

NODE	followpos
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	∅

- Step 4: Construct Dstates, the set of states of DFA D and Dtran, the transition table for D. The start state of DFA D is firstpos(root) and the accepting states are all those containing the position associated with the endmarker symbol #.
- As per example, the firstpos of the root is {1, 2, 3}.
- Let this state be A and consider the input symbol a. Positions 1 and 3 are for a, so let B (the next state) = followpos(1) ∪ followpos(3) = {1, 2, 3, 4}.
- Since this is not in the state list, set Dtran[A, a] := B i.e. {1,2,3,4}.
- Next consider input b, find that out of the positions in A, only 2 is associated with b, so consider the set followpos(2) = {1, 2, 3}.
- As this is already a state, so do not add it to Dstates but add the transition Dtran[A, b]:= A.

- Initial State is the Root ={1,2,3} named as A. To create new states check the follow pos of each input symbol.
- From state A {1,2,3} check follow pos for each symbol in A.
  Here, 1 and 3 represent the same symbol ‘a’ so state is FP(1)UFP(3) ={1,2,3,4} named as B.
  Next FP(2) is {1,2,3} it is already there state A.
- From state B check follow pos of each symbol of state B i.e. {1, 2, 3, 4}
  1 and 3 represent the same symbol ‘a’ so state is FP(1)UFP(3)={1,2,3,4} already there state B.
  2 and 4 represent same symbol ‘b’ so state is FP(2)UFP(4) = {1,2,3,5} name it C.
- From state C ={1,2,3,5}
  1 and 3 represent the same symbol ‘a’ so state is FP(1)UFP(3)={1,2,3,4} already there state B.
  2 and 5 represent same symbol ‘b’ so state is FP(2)UFP(5) = {1,2,3,6} name it D.
- From state D = {1,2,3,6}
  1 and 3 represent same symbol ‘a’ so state is FP(1)UFP(3) = {1,2,3,4} already there state B.
  2 represent symbol ‘b’ so state is FP(2) = {1,2,3} already there state B.
  6 represent symbol ‘#’ so state is FP(6) = ϕ. So D is the final state.

Transition Table

	Input	
State	**a**	**b**
→ A {1,2,3}	B	A
B {1,2,3,4}	B	C
C {1,2,3,5)	B	D
D {1,2,3,6}	B	A

DFA for the transition table