Regular Expression To
DFA

expressions would then have smaller "DFA's" than they do under the standard definition of a DFA. Give an example of one such regular expression.

!! **Exercise 3.8.4:** Design an algorithm to recognize Lex-lookahead patterns of the form r_1/r_2, where r_1 and r_2 are regular expressions. Show how your algorithm works on the following inputs:

a) $(abcd|abc)/d$

b) $(a|ab)/ba$

c) $aa*/a*$

3.9 Optimization of DFA-Based Pattern Matchers

In this section we present three algorithms that have been used to implement and optimize pattern matchers constructed from regular expressions.

1. The first algorithm is useful in a Lex compiler, because it constructs a DFA directly from a regular expression, without constructing an intermediate NFA. The resulting DFA also may have fewer states than the DFA constructed via an NFA.

2. The second algorithm minimizes the number of states of any DFA, by combining states that have the same future behavior. The algorithm itself is quite efficient, running in time $O(n \log n)$, where n is the number of states of the DFA.

3. The third algorithm produces more compact representations of transition tables than the standard, two-dimensional table.

3.9.1 Important States of an NFA

To begin our discussion of how to go directly from a regular expression to a DFA, we must first dissect the NFA construction of Algorithm 3.23 and consider the roles played by various states. We call a state of an NFA *important* if it has a non-ϵ out-transition. Notice that the subset construction (Algorithm 3.20) uses only the important states in a set T when it computes ϵ-*closure*(*move*(T, a)), the set of states reachable from T on input a. That is, the set of states *move*(s, a) is nonempty only if state s is important. During the subset construction, two sets of NFA states can be identified (treated as if they were the same set) if they:

1. Have the same important states, and

2. Either both have accepting states or neither does.

When the NFA is constructed from a regular expression by Algorithm 3.23, we can say more about the important states. The only important states are those introduced as initial states in the basis part for a particular symbol position in the regular expression. That is, each important state corresponds to a particular operand in the regular expression.

The constructed NFA has only one accepting state, but this state, having no out-transitions, is not an important state. By concatenating a unique right endmarker # to a regular expression r, we give the accepting state for r a transition on #, making it an important state of the NFA for $(r)\#$. In other words, by using the *augmented* regular expression $(r)\#$, we can forget about accepting states as the subset construction proceeds; when the construction is complete, any state with a transition on # must be an accepting state.

The important states of the NFA correspond directly to the positions in the regular expression that hold symbols of the alphabet. It is useful, as we shall see, to present the regular expression by its *syntax tree*, where the leaves correspond to operands and the interior nodes correspond to operators. An interior node is called a *cat-node*, *or-node*, or *star-node* if it is labeled by the concatenation operator (dot), union operator |, or star operator $*$, respectively. We can construct a syntax tree for a regular expression just as we did for arithmetic expressions in Section 2.5.1.

Example 3.31 : Figure 3.56 shows the syntax tree for the regular expression of our running example. Cat-nodes are represented by circles. □

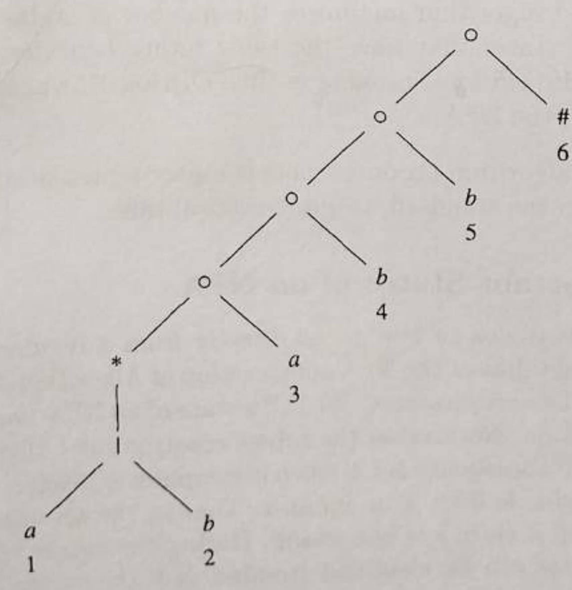


Figure 3.56: Syntax tree for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$

Leaves in a syntax tree are labeled by ϵ or by an alphabet symbol. To each leaf not labeled ϵ, we attach a unique integer. We refer to this integer as the *position* of the leaf and also as a position of its symbol. Note that a symbol can have several positions; for instance, a has positions 1 and 3 in Fig. 3.56. The positions in the syntax tree correspond to the important states of the constructed NFA.

Example 3.32: Figure 3.57 shows the NFA for the same regular expression as Fig. 3.56, with the important states numbered and other states represented by letters. The numbered states in the NFA and the positions in the syntax tree correspond in a way we shall soon see. □

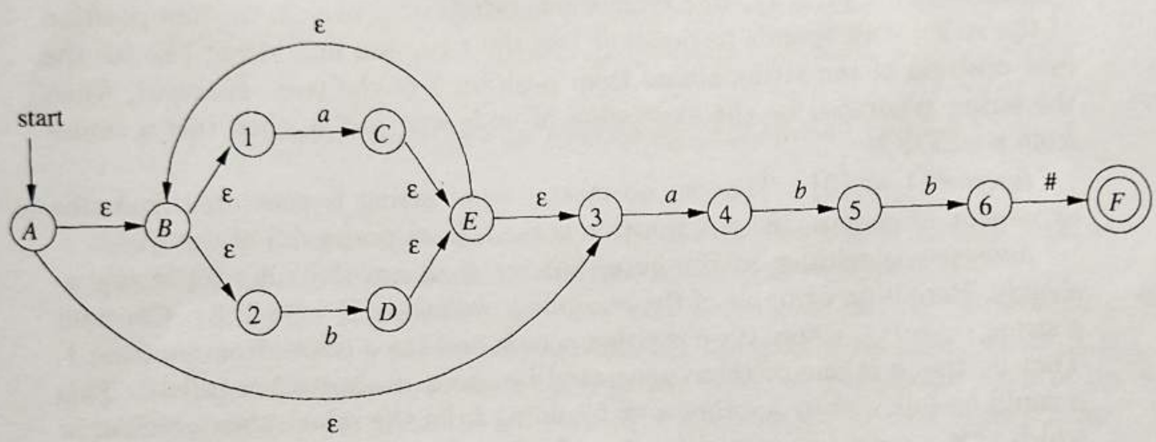


Figure 3.57: NFA constructed by Algorithm 3.23 for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$

3.9.2 Functions Computed From the Syntax Tree

To construct a DFA directly from a regular expression, we construct its syntax tree and then compute four functions: *nullable*, *firstpos*, *lastpos*, and *followpos*, defined as follows. Each definition refers to the syntax tree for a particular augmented regular expression $(r)\#$.

1. $nullable(n)$ is true for a syntax-tree node n if and only if the subexpression represented by n has ϵ in its language. That is, the subexpression can be "made null" or the empty string, even though there may be other strings it can represent as well.

2. $firstpos(n)$ is the set of positions in the subtree rooted at n that correspond to the first symbol of at least one string in the language of the subexpression rooted at n.

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.

4. *followpos*(p), for a position p, is the set of positions q in the entire syntax tree such that there is some string $x = a_1 a_2 \cdots a_n$ in $L\big((r)\#\big)$ such that for some i, there is a way to explain the membership of x in $L\big((r)\#\big)$ by matching a_i to position p of the syntax tree and a_{i+1} to position q.

Example 3.33: Consider the cat-node n in Fig. 3.56 that corresponds to the expression $(\mathbf{a}|\mathbf{b})^*\mathbf{a}$. We claim *nullable*($n$) is false, since this node generates all strings of a's and b's ending in an a; it does not generate ϵ. On the other hand, the star-node below it is nullable; it generates ϵ along with all other strings of a's and b's.

firstpos(n) $= \{1, 2, 3\}$. In a typical generated string like aa, the first position of the string corresponds to position 1 of the tree, and in a string like ba, the first position of the string comes from position 2 of the tree. However, when the string generated by the expression of node n is just a, then this a comes from position 3.

lastpos(n) $= \{3\}$. That is, no matter what string is generated from the expression of node n, the last position is the a from position 3 of the tree.

followpos is trickier to compute, but we shall see the rules for doing so shortly. Here is an example of the reasoning: *followpos*(1) $= \{1, 2, 3\}$. Consider a string $\cdots ac \cdots$, where the c is either a or b, and the a comes from position 1. That is, this a is one of those generated by the $\mathbf{a}$ in expression $(\mathbf{a}|\mathbf{b})^*$. This a could be followed by another a or b coming from the same subexpression, in which case c comes from position 1 or 2. It is also possible that this a is the last in the string generated by $(\mathbf{a}|\mathbf{b})^*$, in which case the symbol c must be the a that comes from position 3. Thus, 1, 2, and 3 are exactly the positions that can follow position 1. □

3.9.3 Computing *nullable*, *firstpos*, and *lastpos*

We can compute *nullable*, *firstpos*, and *lastpos* by a straightforward recursion on the height of the tree. The basis and inductive rules for *nullable* and *firstpos* are summarized in Fig. 3.58. The rules for *lastpos* are essentially the same as for *firstpos*, but the roles of children c_1 and c_2 must be swapped in the rule for a cat-node.

Example 3.34: Of all the nodes in Fig. 3.56 only the star-node is nullable. We note from the table of Fig. 3.58 that none of the leaves are nullable, because they each correspond to non-ϵ operands. The or-node is not nullable, because neither of its children is. The star-node is nullable, because every star-node is nullable. Finally, each of the cat-nodes, having at least one nonnullable child, is not nullable.

The computation of *firstpos* and *lastpos* for each of the nodes is shown in Fig. 3.59, with *firstpos*(n) to the left of node n, and *lastpos*(n) to its right. Each of the leaves has only itself for *firstpos* and *lastpos*, as required by the rule for non-ϵ leaves in Fig. 3.58. For the or-node, we take the union of *firstpos* at the

NODE n	*nullable*(n)	*firstpos*(n)
A leaf labeled ϵ	**true**	$\emptyset$
A leaf with position i	**false**	$\{i\}$
An or-node $n = c_1 \mid c_2$	*nullable*(c_1) **or** *nullable*(c_2)	*firstpos*(c_1) $\cup$ *firstpos*(c_2)
A cat-node $n = c_1c_2$	*nullable*(c_1) **and** *nullable*(c_2)	**if** (*nullable*(c_1)) *firstpos*(c_1) $\cup$ *firstpos*(c_2) **else** *firstpos*(c_1)
A star-node $n = c_1{}^*$	**true**	*firstpos*(c_1)

Figure 3.58: Rules for computing *nullable* and *firstpos*

children and do the same for *lastpos*. The rule for the star-node says that we take the value of *firstpos* or *lastpos* at the one child of that node.

Now, consider the lowest cat-node, which we shall call n. To compute *firstpos*(n), we first consider whether the left operand is nullable, which it is in this case. Therefore, *firstpos* for n is the union of *firstpos* for each of its children, that is $\{1,2\} \cup \{3\} = \{1,2,3\}$. The rule for *lastpos* does not appear explicitly in Fig. 3.58, but as we mentioned, the rules are the same as for *firstpos*, with the children interchanged. That is, to compute *lastpos*(n) we must ask whether its right child (the leaf with position 3) is nullable, which it is not. Therefore, *lastpos*(n) is the same as *lastpos* of the right child, or $\{3\}$. □

3.9.4 Computing *followpos*

Finally, we need to see how to compute *followpos*. There are only two ways that a position of a regular expression can be made to follow another.

1. If n is a cat-node with left child c_1 and right child c_2, then for every position i in *lastpos*(c_1), all positions in *firstpos*(c_2) 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).

Example 3.35 : Let us continue with our running example; recall that *firstpos* and *lastpos* were computed in Fig. 3.59. Rule 1 for *followpos* requires that we look at each cat-node, and put each position in *firstpos* of its right child in *followpos* for each position in *lastpos* of its left child. For the lowest cat-node in Fig. 3.59, that rule says position 3 is in *followpos*(1) and *followpos*(2). The next cat-node above says that 4 is in *followpos*(3), and the remaining two cat-nodes give us 5 in *followpos*(4) and 6 in *followpos*(5).

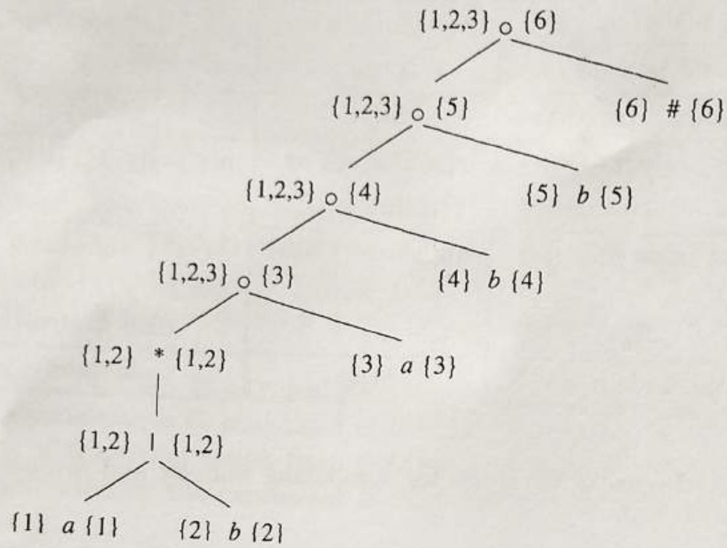


Figure 3.59: *firstpos* and *lastpos* for nodes in the syntax tree for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb\#}$

We must also apply rule 2 to the star-node. That rule tells us positions 1 and 2 are in both *followpos*(1) and *followpos*(2), since both *firstpos* and *lastpos* for this node are $\{1, 2\}$. The complete sets *followpos* are summarized in Fig. 3.60. □

NODE n	*followpos*(n)
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

Figure 3.60: The function *followpos*

We can represent the function *followpos* by creating a directed graph with a node for each position and an arc from position i to position j if and only if j is in *followpos*(i). Figure 3.61 shows this graph for the function of Fig. 3.60.

It should come as no surprise that the graph for *followpos* is almost an NFA without ϵ-transitions for the underlying regular expression, and would become one if we:

1. Make all positions in *firstpos* of the root be initial states,

2. Label each arc from i to j by the symbol at position i, and

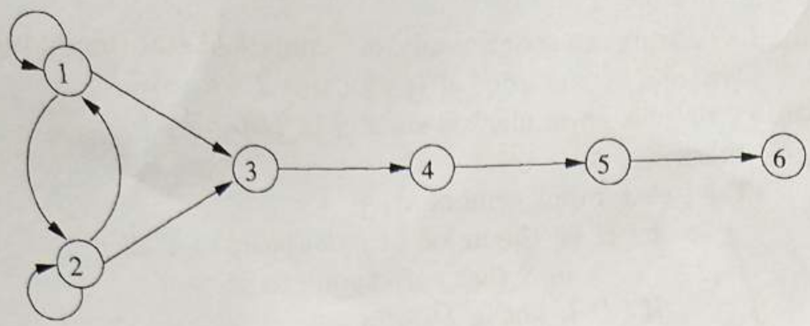


Figure 3.61: Directed graph for the function *followpos*

3. Make the position associated with endmarker # be the only accepting state.

3.9.5 Converting a Regular Expression Directly to a DFA

Algorithm 3.36: Construction of a DFA from a regular expression r.

INPUT: A regular expression r.

OUTPUT: A DFA D that recognizes $L(r)$.

METHOD:

1. Construct a syntax tree T from the augmented regular expression $(r)\#$.
2. Compute *nullable*, *firstpos*, *lastpos*, and *followpos* for T, using the methods of Sections 3.9.3 and 3.9.4.
3. Construct *Dstates*, the set of states of DFA D, and *Dtran*, the transition function for D, by the procedure of Fig. 3.62. The states of D are sets of positions in T. Initially, each state is "unmarked," and a state becomes "marked" just before we consider its out-transitions. The start state of D is $\mathit{firstpos}(n_0)$, where node n_0 is the root of T. The accepting states are those containing the position for the endmarker symbol #.

□

Example 3.37: We can now put together the steps of our running example to construct a DFA for the regular expression $r = (\mathbf{a}|\mathbf{b})^*\mathbf{abb}$. The syntax tree for $(r)\#$ appeared in Fig. 3.56. We observed that for this tree, *nullable* is true only for the star-node, and we exhibited *firstpos* and *lastpos* in Fig. 3.59. The values of *followpos* appear in Fig. 3.60.

The value of *firstpos* for the root of the tree is $\{1, 2, 3\}$, so this set is the start state of D. Call this set of states A. We must compute $\mathit{Dtran}[A, a]$ and $\mathit{Dtran}[A, b]$. Among the positions of A, 1 and 3 correspond to a, while 2 corresponds to b. Thus, $\mathit{Dtran}[A, a] = \mathit{followpos}(1) \cup \mathit{followpos}(3) = \{1, 2, 3, 4\}$,

initialize *Dstates* to contain only the unmarked state *firstpos*(n_0),
    where n_0 is the root of syntax tree T for $(r)\#$;
while (there is an unmarked state S in *Dstates*) {
    mark S;
    for (each input symbol a) {
        let U be the union of *followpos*(p) for all p
            in S that correspond to a;
        if (U is not in *Dstates*)
            add U as an unmarked state to *Dstates*;
        Dtran$[S, a] = U$;
    }
}

Figure 3.62: Construction of a DFA directly from a regular expression

and *Dtran*$[A, b] =$ *followpos*$(2) = \{1, 2, 3\}$. The latter is state A, and so does not have to be added to *Dstates*, but the former, $B = \{1, 2, 3, 4\}$, is new, so we add it to *Dstates* and proceed to compute its transitions. The complete DFA is shown in Fig. 3.63. □

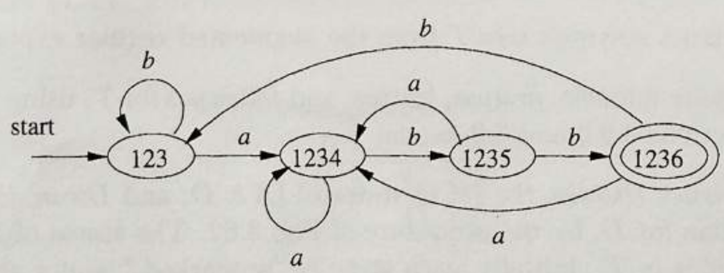


Figure 3.63: DFA constructed from Fig. 3.57

3.9.6 Minimizing the Number of States of a DFA

There can be many DFA's that recognize the same language. For instance, note that the DFA's of Figs. 3.36 and 3.63 both recognize language $L((\mathbf{a}|\mathbf{b})^*\mathbf{abb})$. Not only do these automata have states with different names, but they don't even have the same number of states. If we implement a lexical analyzer as a DFA, we would generally prefer a DFA with as few states as possible, since each state requires entries in the table that describes the lexical analyzer.

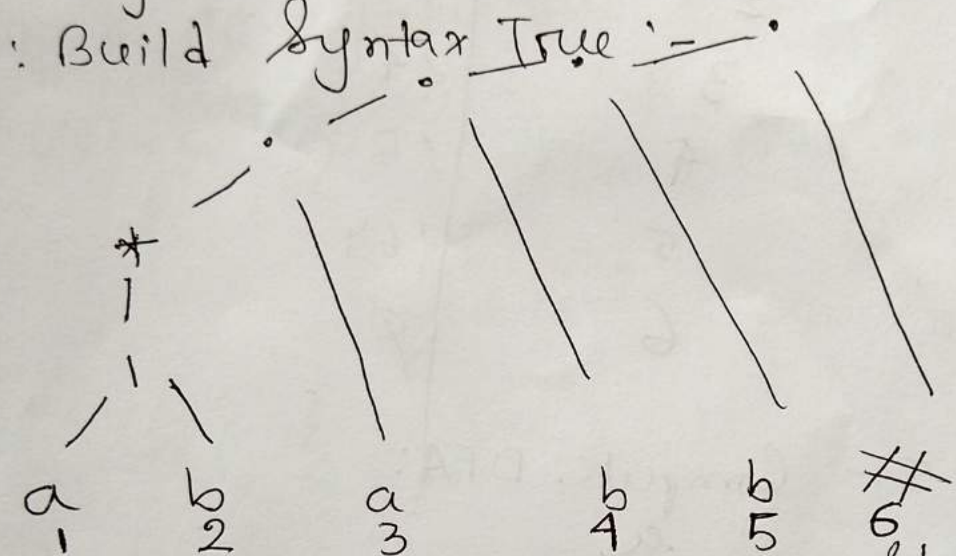
The matter of the names of states is minor. We shall say that two automata are *the same up to state names* if one can be transformed into the other by doing nothing more than changing the names of states. Figures 3.36 and 3.63 are not the same up to state names. However, there is a close relationship between the

Q1. Convert the following regular expression directly to DFA: $(a|b)^*abb$

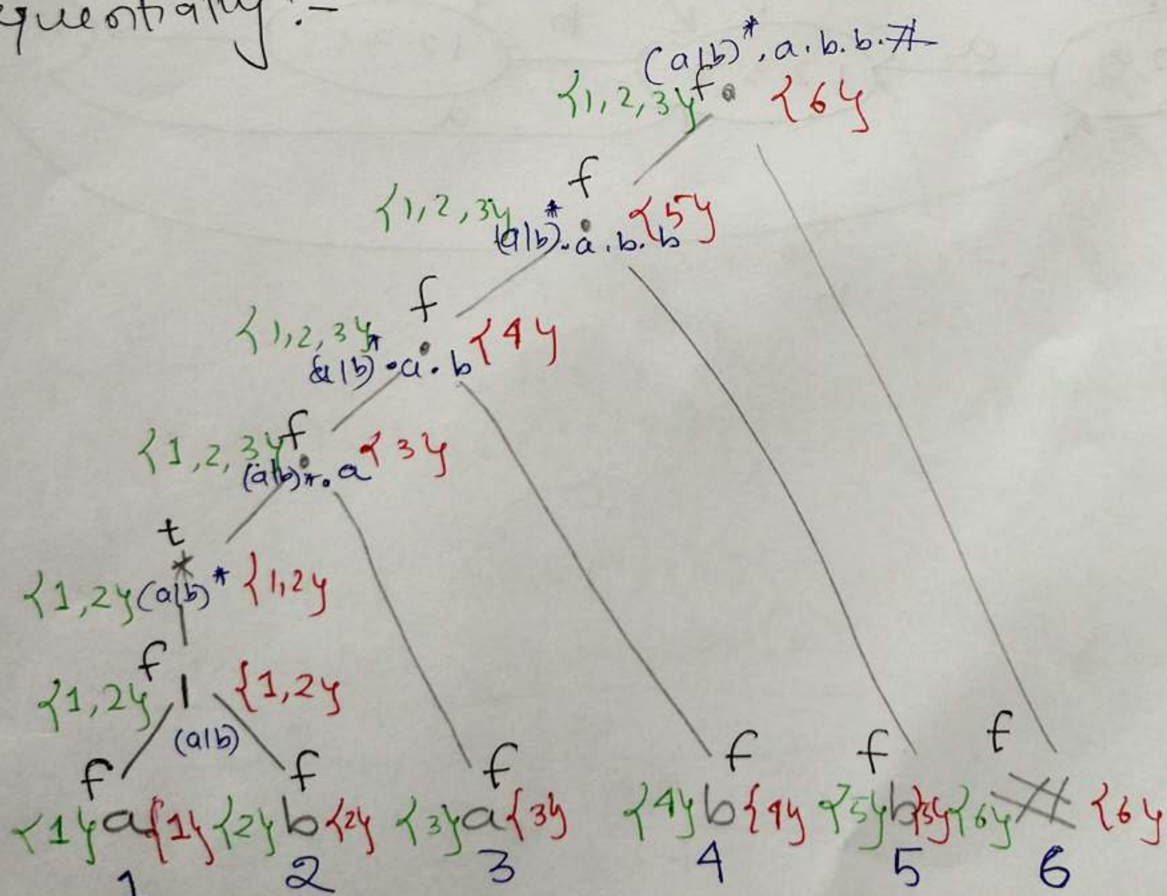
Solution:-

Step 1: Augment the RE: $(a|b)^*.a.b.b.\#$

Step 2: Build Syntax Tree:-



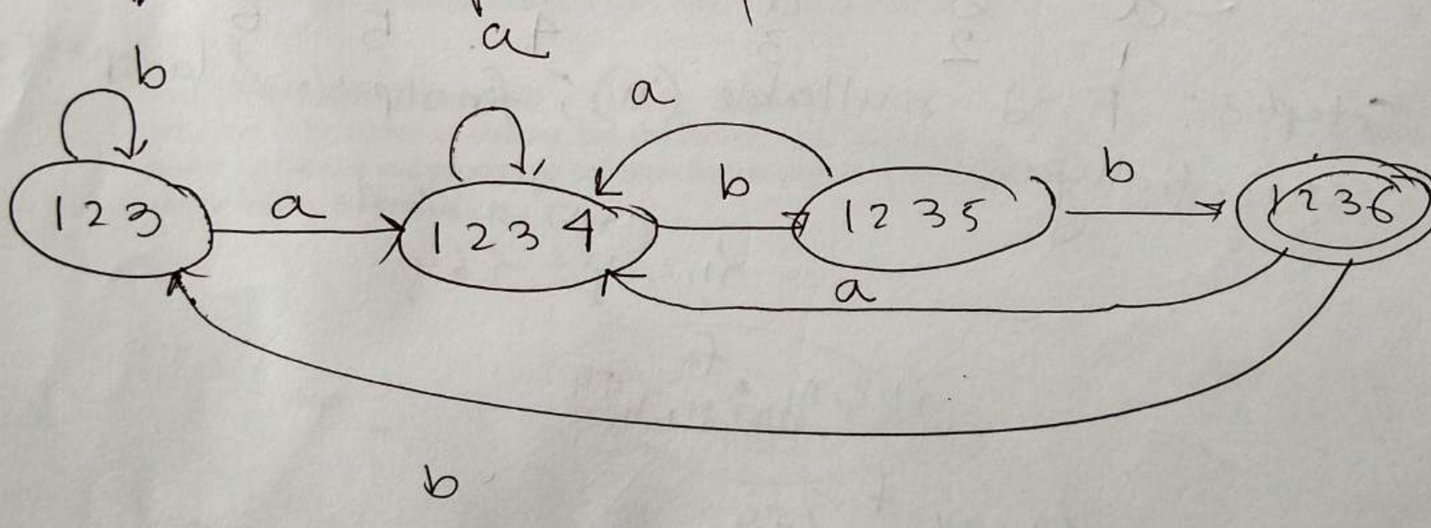
Step 3: Find nullable(n), firstpos(n) & lastpos(n) sequentially:-



Step 4: Compute followpos(n)

Node (n)	followpos (n)
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	ϕ

Step 5:- Compute DFA:



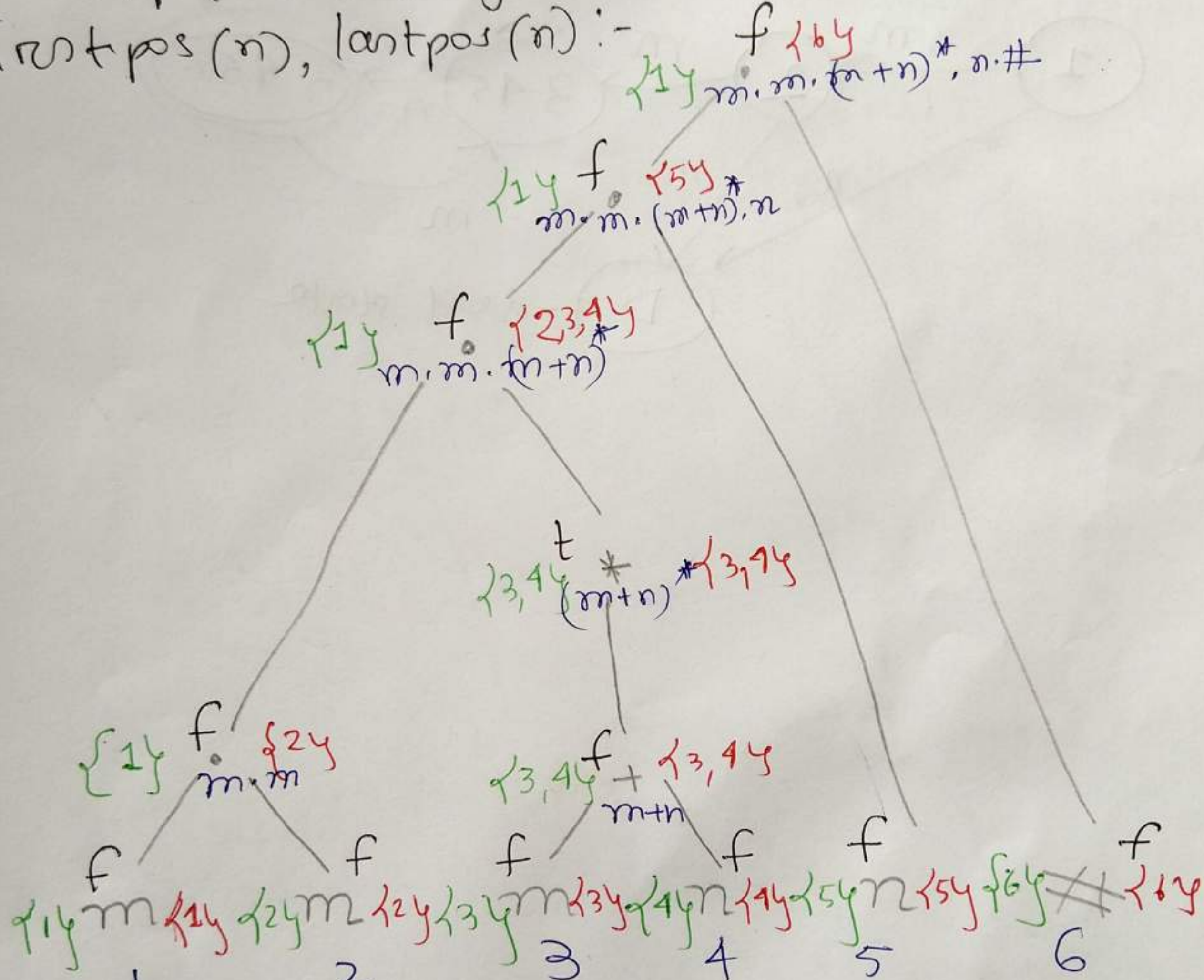
Q2. Convert the following RE directly to DFA:

$m\,m\,(m+n)^*\,n$

Solution:-

Step 1 : Augment the RE : $m\,m\,(m+n)^*\,n\,\#$

Step 2 : Build syntax tree, find nullable(n), firstpos(n), lastpos(n) :-

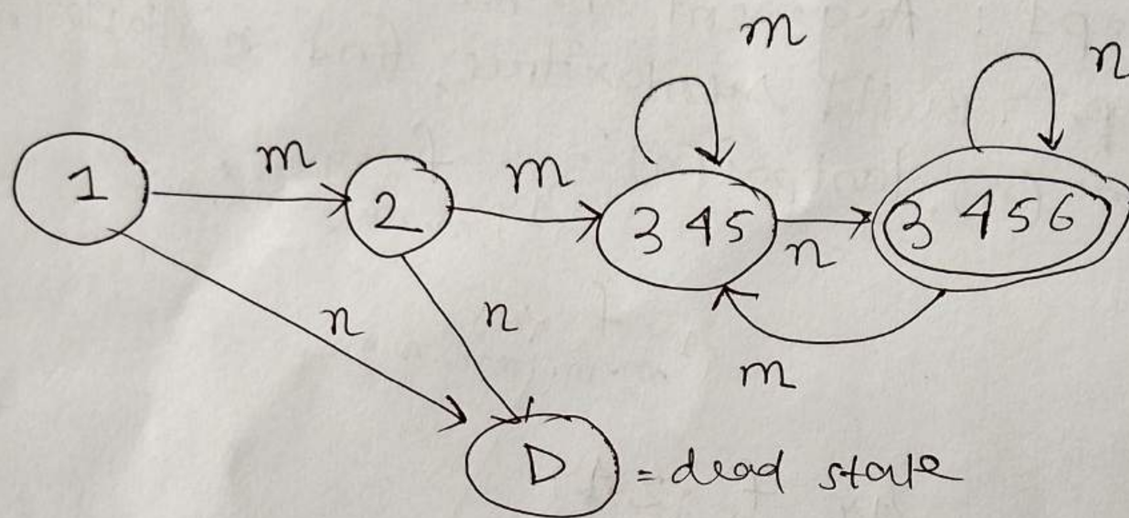


Step 3 : find followpos(n):

Node (n)	followpos (n)
1	2
2	3,4,5
3	~~5~~ 3,4,5
4	~~5~~ 3,4,5
5	6
6	∅

step 4: DFA:

first pos (root) = {1}



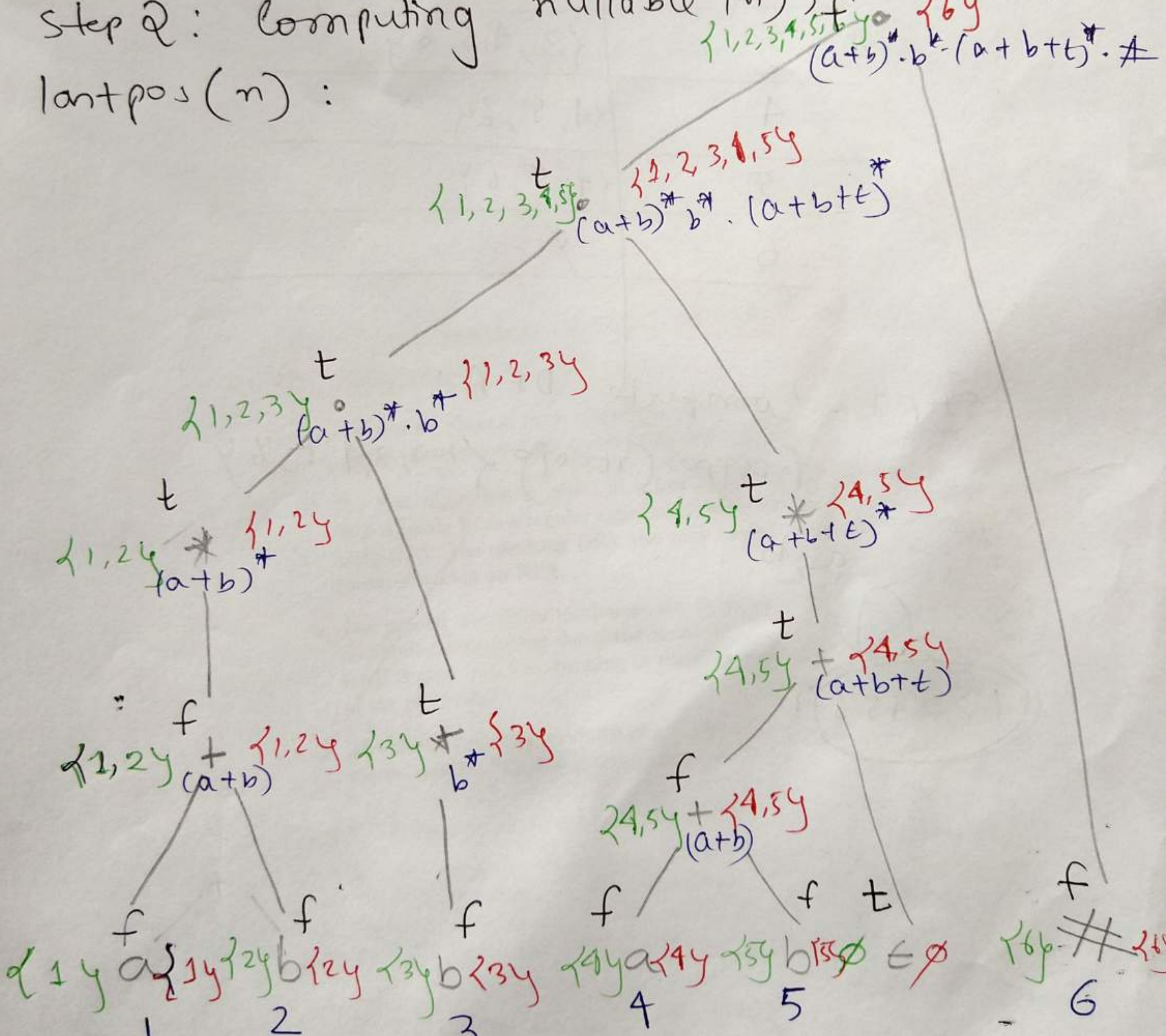
Convert the RE directly to DFA: ⑤

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

Solution:-

Step 1: Augmented RE: $(a+b)^* b^* (a+b+t)^* \#$

Step 2: Computing nullable(n), firstpos(n), lastpos(n):



step 3: Compute followpos(n):

Node (n)	followpos (n)
1	{1, 2, 3, 4, 5, 6}
2	{1, 2, 3, 4, 5, 6}
3	{3, 4, 5, 6}
4	{4, 5, 6}
5	{4, 5, 6}
6	ϕ

step 4: Compute DFA:

firstpos (root) = {1, 2, 3, 4, 5, 6}