BCSE307L – COMPILER DESIGN

TEXT BOOK:

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques and Tools", Second Edition, Pearson Education Limited, 2014.

Module:1	INTRODUCTION TO COMPILATION AND LEXICAL ANALYSIS	7 hours
Introduction to LLVM - Structure and Phases of a Compiler-Design Issues-Patterns-Lexemes-Tokens-Attributes-Specification of Tokens-Extended Regular Expression- Regular expression to Deterministic Finite Automata (Direct method) - Lex - A Lexical Analyzer Generator.		

RE to DFA

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 $\epsilon\text{-}closure(move(T, a))$, 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.

RE to DFA

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. □

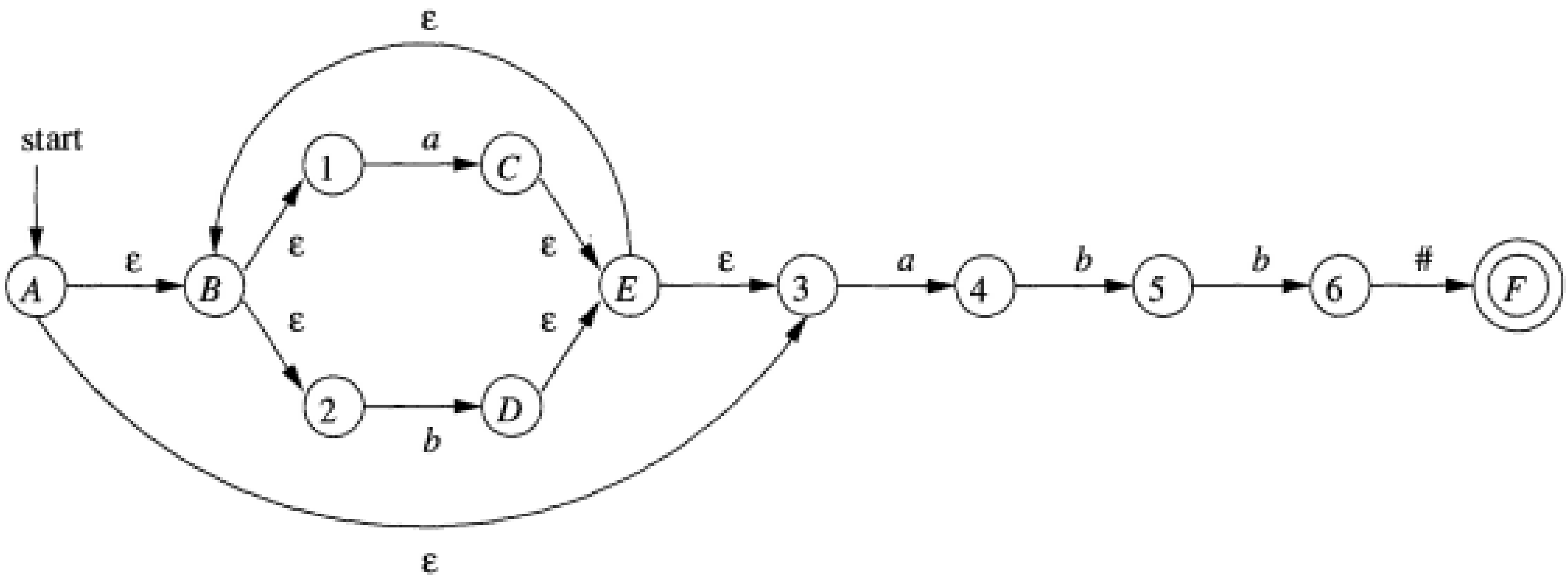


Figure 3.57: NFA constructed by Algorithm 3.23 for (**a**|**b**)***abb#**

1. Syntax Tree

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

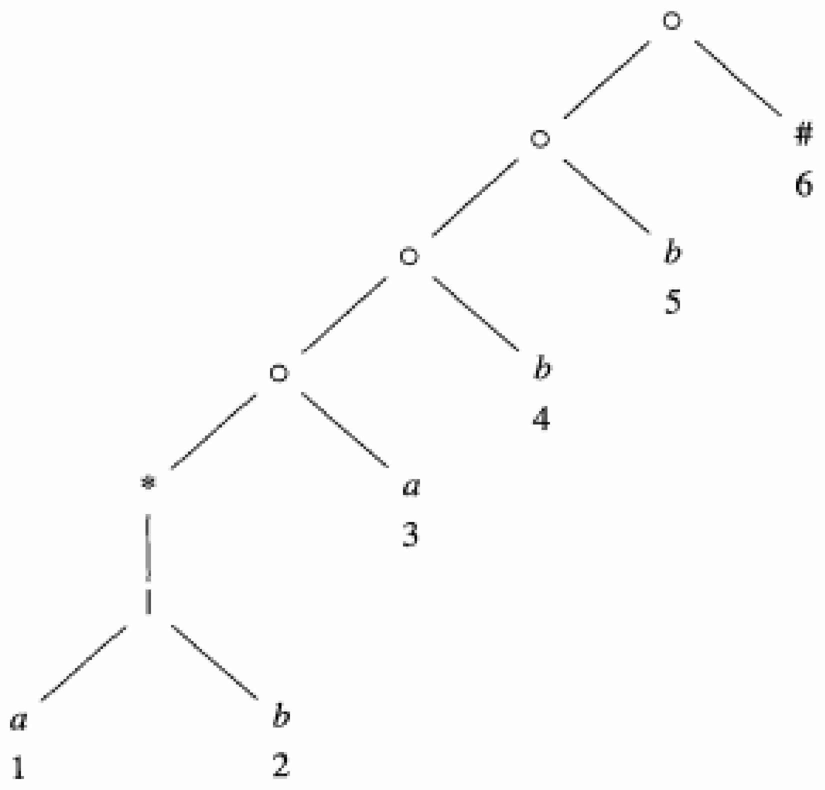


Figure 3.56: Syntax tree for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb\#}$

1. Syntax Tree

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.

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 $x = a_1a_2\cdots a_n$ 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.

2. Compute Nullable, Firstpos, Lastpos

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*

2. Compute Nullable, Firstpos, Lastpos

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 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 $\{3\}$.

2. Compute Nullable, Firstpos, Lastpos

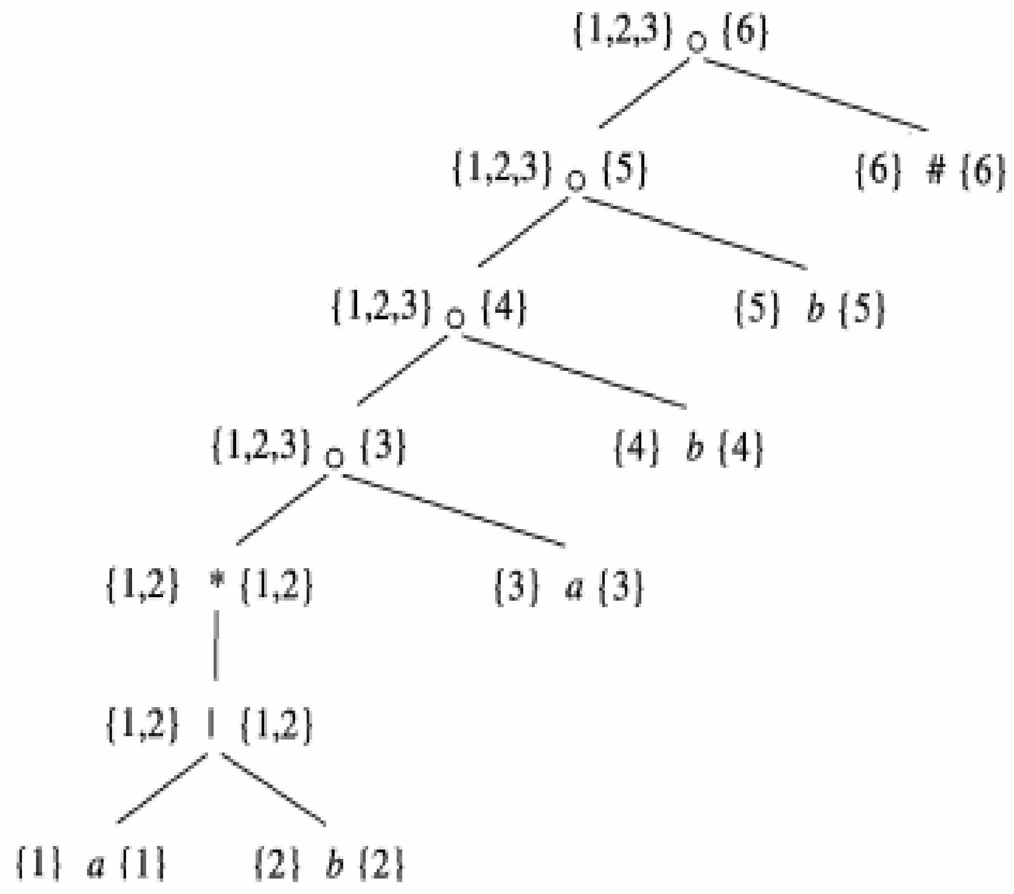


Figure 3.59: *firstpos* and *lastpos* for nodes in the syntax tree for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb\#}$

3. Compute Followpos

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 $followpos(5)$.

3. Compute Followpos

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*

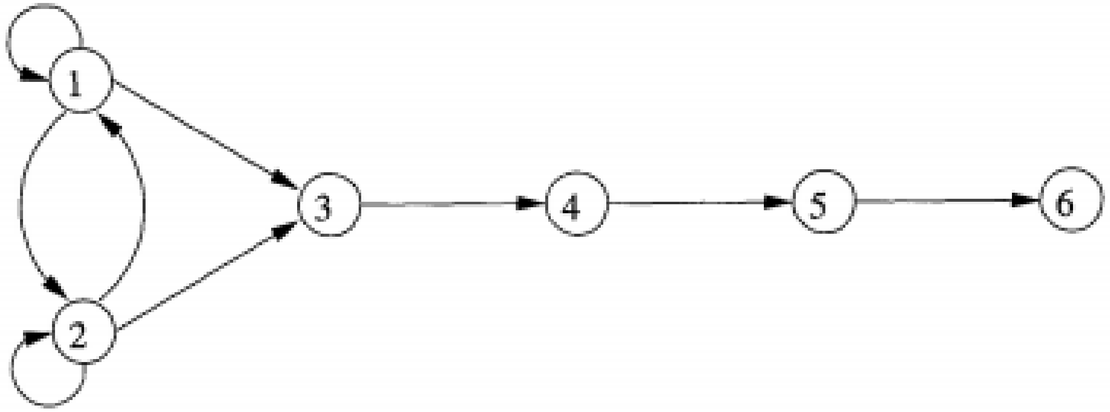


Figure 3.61: Directed graph for the function *followpos*

4. Dtrans - RE to DFA (Direct Method)

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 $firstpos(n_0)$, where node n_0 is the root of T. The accepting states are those containing the position for the endmarker symbol $\#$.

4. Dtrans - RE to DFA (Direct Method)

```
initialize Dstates to contain only the unmarked state firstpos(n0),
        where n0 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

4. Dtrans - RE to DFA (Direct Method)

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 $Dtran[A, a]$ and $Dtran[A, b]$. Among the positions of A, 1 and 3 correspond to a, while 2 corresponds to b. Thus, $Dtran[A, a] = followpos(1) \cup followpos(3) = \{1, 2, 3, 4\}$,

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. □

iv) Construction of DFA:

$A = \{1, 2, 3\}$

D-trans [mov (A, a)] = followpos(1) ∪ followpos(3) = $\{1, 2, 3, 4\} \Rightarrow B$

D-trans [mov (A, b)] = followpos(2) = $\{1, 2, 3\} \Rightarrow A$

D-trans [mov (B, a)] = followpos(1) ∪ followpos(3) = $\{1, 2, 3, 4\} \Rightarrow B$

D-trans [mov (B, b)] = followpos(2) ∪ followpos(4) = $\{1, 2, 3, 5\} \Rightarrow C$

D-trans [mov (C, a)] = followpos(1) ∪ followpos(3) = $\{1, 2, 3, 4\} \Rightarrow B$

D-trans [mov (C, b)] = followpos(2) ∪ followpos(5) = $\{1, 2, 3, 6\} \Rightarrow D$

D-trans [mov (D, a)] = followpos(1) ∪ followpos(3) = $\{1, 2, 3, 4\} \Rightarrow B$

D-trans [mov (D, b)] = followpos(2) = $\{1, 2, 3\} \Rightarrow A$

Transition Table for DFA:

States	a	b
A	B	A
B	B	C
C	B	D
D	B	A

DFA Transition Diagram

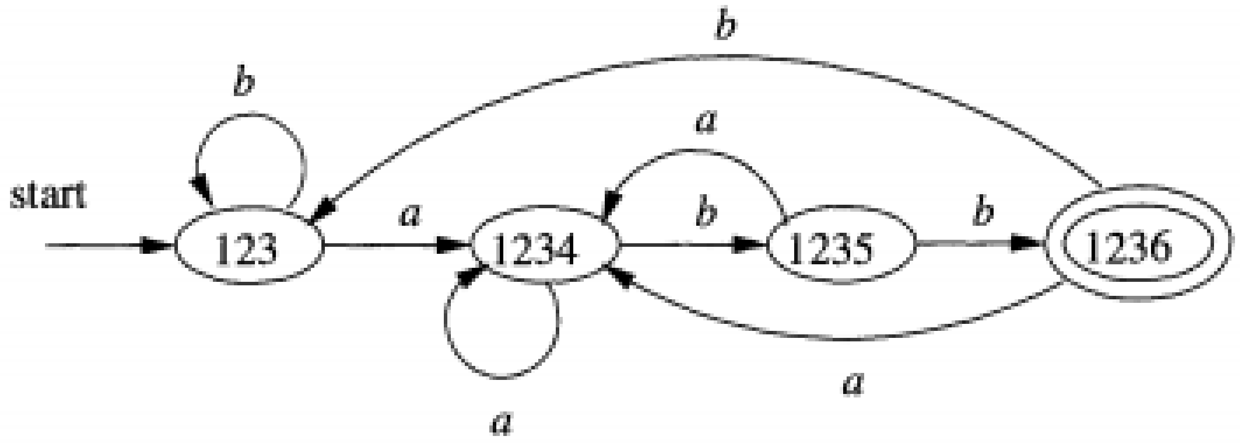
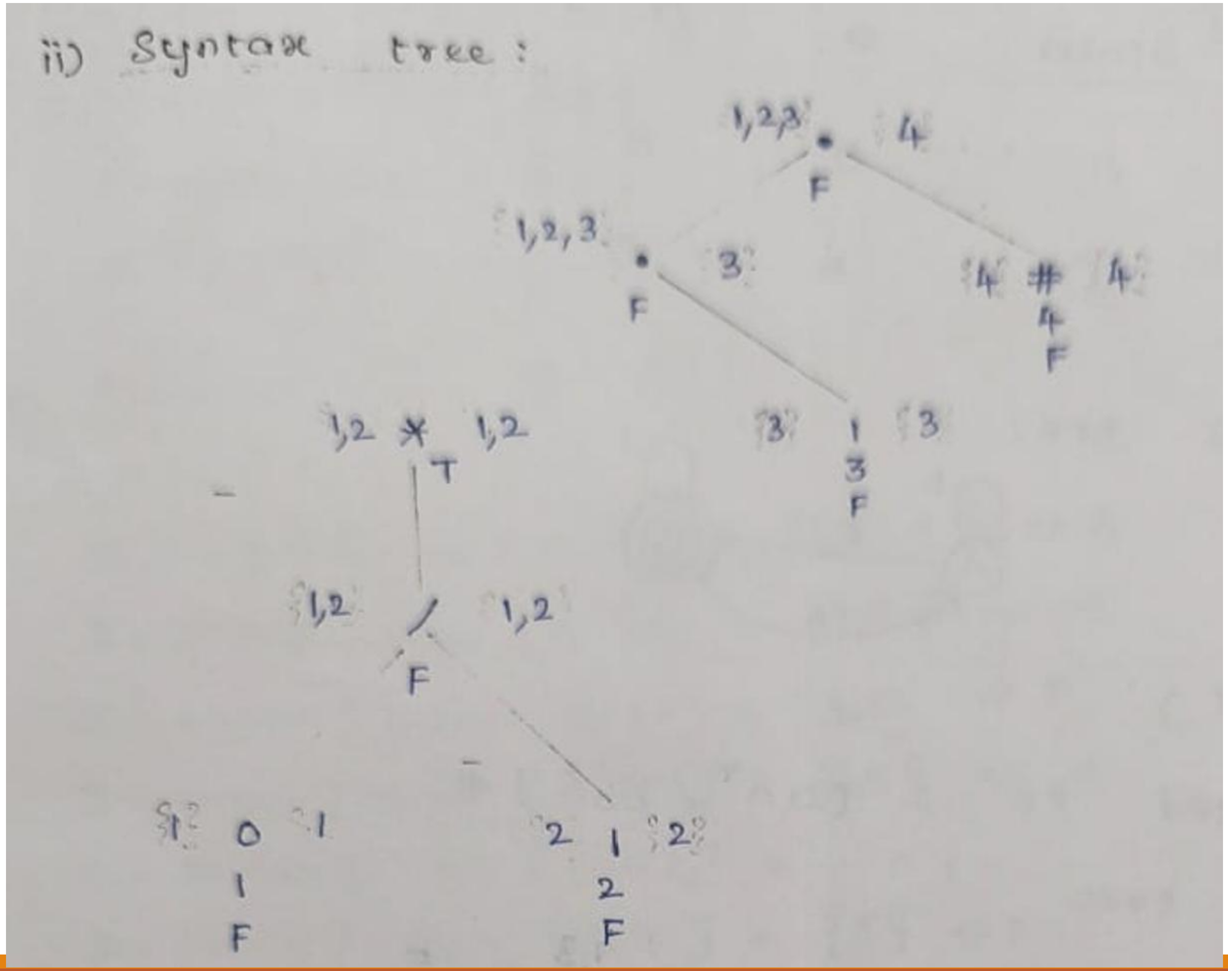


Figure 3.63: DFA constructed from Fig. 3.57

Example 2: (0|1)*1#
Syntax Tree , First Position & Last Position



Followpos (1) = {1, 2, 3}
Followpos (2) = {1, 2, 3}
Followpos (3) = {4}
Followpos (4) = -

v) Construction of DFA:

A = {1, 2, 3}

D-trans[Mov(A, 0)] = followpos(1)
= {1, 2, 3} ⇒ A

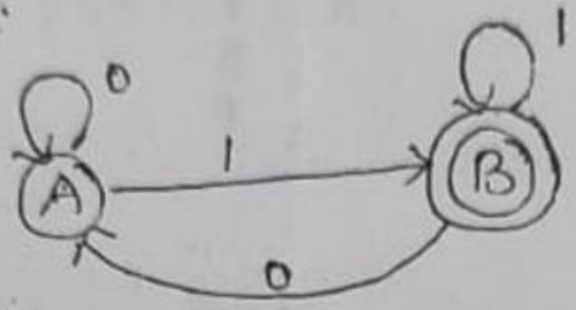
D-trans['Mov(A, 1)] = followpos(2) ∪ followpos(3)
= {1, 2, 3, 4} ⇒ B

D-trans[Mov(B, 0)] = {1, 2, 3} ⇒ A
D-trans[Mov(B, 1)] = {1, 2, 3, 4} ⇒ B

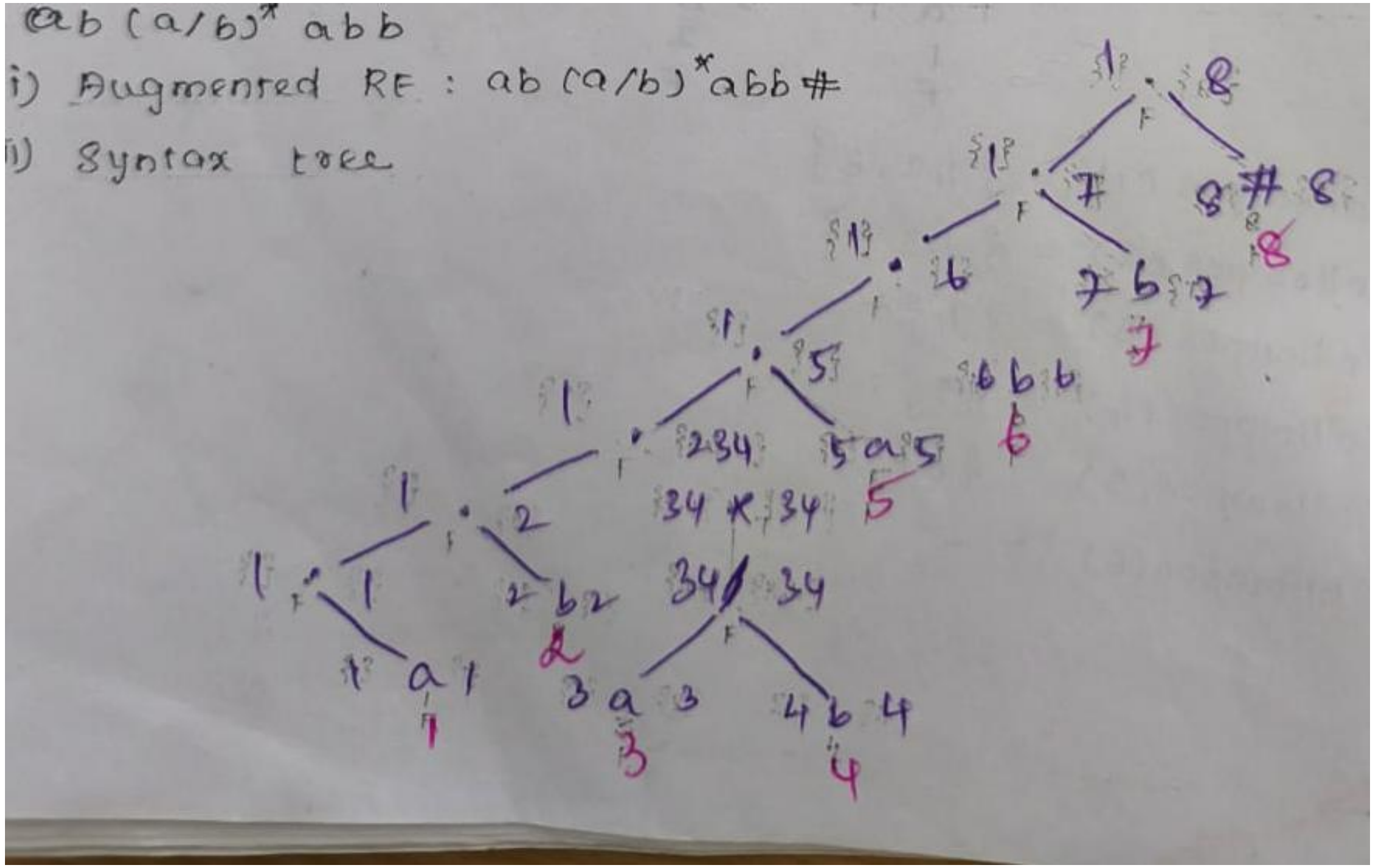
Transition Table:

States	Input 0	Input 1
A	A	B
B	A	B

Minimized DFA:



Example 3: ab(a|b)*abb#
Syntax Tree , First Position & Last Position



Follow position

Followpos (1) = {2}

Followpos (2) = {3, 4, 5}

Followpos (3) = {3, 4, 5}

Followpos (4) = {3, 4, 5}

Followpos (5) = {6}

Followpos (6) = {7}

Followpos (7) = {8}

Followpos (8) = –

Construction of DFA:

A = {1}

D-trans[mov(A, a)] = followpos(1) = {2} ⇒ B

D-trans[mov(A, b)] = –

D-trans[mov(B, a)] = –

D-trans[mov(B, b)] = {3, 4, 5} ⇒ C

D-trans[mov(C, a)] = {3, 4, 5, 6} ⇒ D

D-trans[mov(C, b)] = {3, 4, 5} ⇒ C

D-trans[mov(D, a)] = {3, 4, 5, 6} ⇒ D

D-trans[mov(D, b)] = {3, 4, 5, 7} ⇒ E

D-trans[mov(E, a)] = {3, 4, 5, 6} ⇒ D

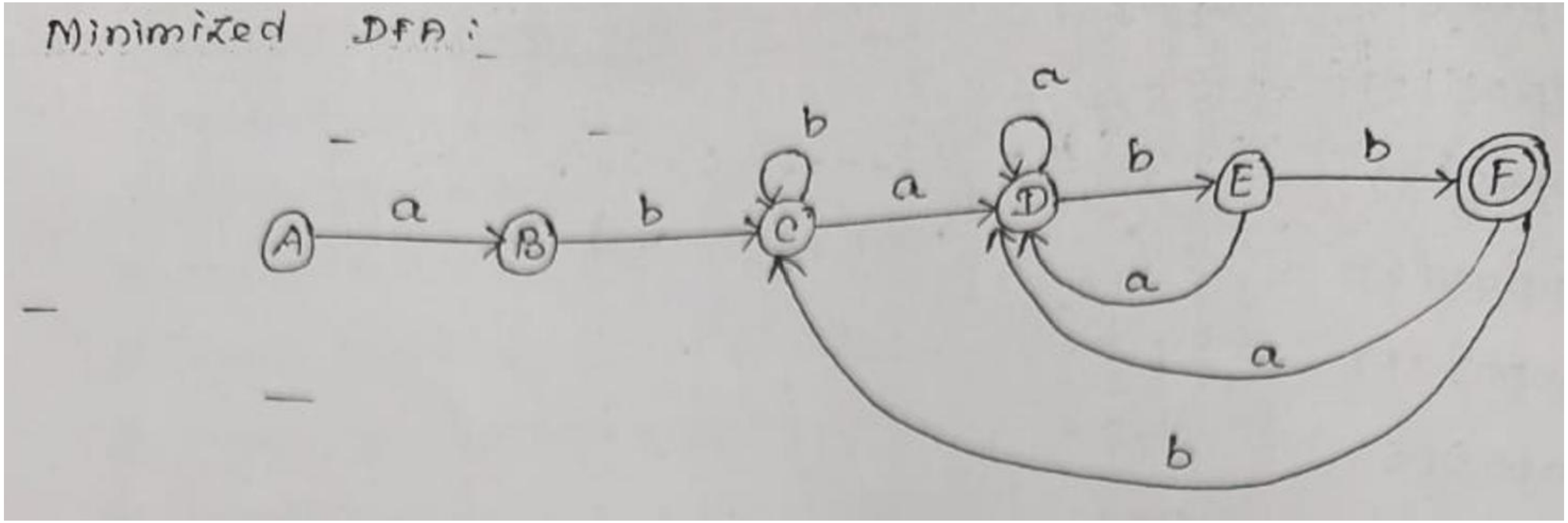
D-trans[mov(E, b)] = {3, 4, 5, 8} ⇒ F

D-trans[mov(F, a)] = {3, 4, 5, 6} ⇒ D

D-trans[mov(F, b)] = {3, 4, 5} ⇒ C

Transition Table for DFA:

States	a	b
A	B	—
B	—	C
C	D	C
D	D	E
E	D	F
F	D	C


Minimized DFA :
a
b
b
a
a
b
b
a
a
b
A
B
C
D
E
F

Thank You