Module 6 – Lexical Phase - RE to DFA

The objective of this module is to construct a minimized DFA from a regular expression. A NFA is typically easier to construct but string matching with a NFA is slower. Hence, we go in for a DFA representation. However, if the regular expression is converted to a DFA using the Thompsonøs subset construction algorithm, which was discussed in modules 4 and 5, the resultant DFA will have more states than actually necessary. Therefore for string matching, this DFA would take more than the optimized DFA that is directly constructed for the language. So, in this module, we will discuss the construction of a minimized DFA from a regular expression.

6.1 Minimized DFA Construction

A DFA can be constructed for a language using a direct representation as a directed graph, or as a procedure that converts a regular expression to a DFA. The conversion from RE to DFA can be done using the following procedure

- Thompsonøs subset construction algorithm ó results in a non-minimized DFA and hence could be minimized using Table filling algorithm
- Syntax tree procedure ó Results in a minimized DFA

In this module, the syntax tree procedure that converts a regular expression to a minimized DFA is discussed.

6.2 Syntax Tree Procedure

The following algorithm, gives the steps to be followed in converting the regular expression (RE) to a DFA.

SyntaxTreeAlgorithm

Input: Regular Expression

Output: a Minimized DFA

1. Augment the regular expression ***r*** with a special symbol # which is used as an end marker and any transition over # in a DFA will be an accepting. Hence, the new expression is ***r* #.**

2. Construct a syntax tree for ***r#***

3. Traverse the tree to construct functions *nullable( )*, *firstpos( )*, *lastpos ( )*, and *followpos( )*

4. Based on the functions *firstpos( )*, *lastpos ( )*, and *followpos( )* and using the tree, the minimized DFA is constructed

   Let us discuss each of these functions in detail before going ahead with the implementation of the algorithm:

e-Pathshala पाठशाला A Gateway to All Post Graduate Courses

6.2.1 Syntax Tree construction

Based on the precedence of the regular expression operators õ*ö, õ+ö and õ.ö a syntax tree is constructed. The kleene closure operator * has the highest precedence and will have just one child. The next precedence is assigned to the concatenation operator õ.ö and this will have two children. The least precedence is given to the union operator õ+ö and this will also have two children. The input symbols and the special symbol õ#ö will be the leaf nodes of this expression.

If õaö and õbö are input symbols and they are connected using the õ+ö operator then the construction of the syntax tree is given in figure 6.1 (a). The syntax tree will be a typical binary tree.

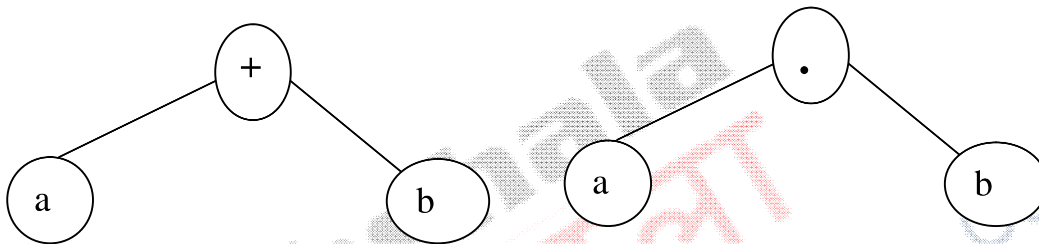

Figure 6.1(a) Syntax tree for a+b **Figure 6.1 (b) Syntax tree for a . b**

If the input symbols are connected by the concatenation operator then the syntax tree would be as given in figure 6.1 (b). If the input symbol has a kleene closure operator, then the syntax tree would be as given in figure 6.1 (c)

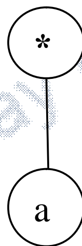

Figure 6.1(c) Syntax tree for a*

Using this method and the precedence of operators, a syntax tree for the regular expression r# is constructed. All the leaf nodes are labeled with integers from 1 to n, which would be used as the information to construct the DFA later.

6.2.2. Nullable( )

After constructing the syntax tree, for every symbol in the syntax tree, the function nullable() is defined. The leaf nodes are first assigned nullable based on whether the sub-tree under them can generate an empty string . Hence, if a leaf node is labeled with õ ö then the nullable information

ePathshala पाठशाला A Gateway to All Post Graduate Courses

of that node is set to õTrueö and for all other input alphabet õaö such that õaö is not an operator, the nullable information of that node is set to õFalseö. For the regular expression operators, nullable information is assigned. They form the interior node part of the syntax tree. The concatenation operator and the union operator cannot generate empty string and hence their nullable information depends on the nullable status of its children. The concatenation operator is considered as õandö operation while the union operator is considered as õorö operation for determining the nullable information of the interior node. If c1 and c2 are the children of these interior nodes, then the following relationship is used to calculate the nullable information of the interior nodes õ+ö and õ.ö

nullable (+) = nullable(c1) or nullable(c2) (6.1)

nullable (.) = nullable (c1) and nullable(c2) (6.2)

Here the operators õorö and õandö indicates the logical operators. Hence, nullable(+) will be set to õTrueö if any of its children is nullable and nullable(.) will be set to õTrueö only if both of its children are nullable. Since, the kleene closure operator can generate as a string, the nullable of the * node is set to õTrueö

nullabe(*) = True (6.3)

6.2.3 firstpos()

The function firstpos(n) is defined as the set of positions that can match the first symbol of a string generated by the sub-tree at node õnö. It is defined for all the nodes and depends on the firstpos() values of the leaf nodes.

To start with, for any input symbol õaö, the firstpos(a) is the integer assigned to it. For the node õ ö, firstpos( ) is defined as empty. After assigning the firstpos( ) for all the leaf nodes, firstpos() of all interior nodes are computed. To compute this, nullable information of all interior and leaf nodes are necessary. For a node labeled with the Kleene closure operator, *, and child c1 the firstpos() is defined as

firstpos(*) = firstpos(c1) (6.4)

For interior nodes labeled with õ+ö and õ.öwith children c1 and c2, firstpos() is defined as follows:

firstpos(+) = firstpos(c1) U firstpos(c2) (6.5)

firstpos(.) = if (nullable(c1) == true) then
return (firstpos(c1) $\cup$ firstpos(c2))

ePathshala पाठशाला A Gateway to All Post Graduate Courses

else
return firstpos(c1) (6.6)

This function is computed for all interior and leaf nodes and the result is stored in a list.

6.2.4 lastpos()

lastpos() is also computed for all the nodes of the syntax tree. lastpos(n) is defined as the set of positions that can match the last symbol of a string generated be the subtree at node *n.*

To start with, for any input symbol õaö, the lastpos(a) is also the integer assigned to it similar to the firstpos(). For the node õ ö, lastpos( ) is defined as empty, since which is defined as the suffix or prefix of any string is not shown with the string during representation. After assigning the lastpos( ) for all the leaf nodes, lastpos() of all interior nodes are computed. To compute this, nullable information of all interior and leaf nodes is necessary. For a node labeled with the Kleene closure operator, *, and child c1 the lastpos() is defined as

lastpos(*) = lastpos(c1) (6.7)

For interior nodes labeled õ+ö and õ.ö and with children c1 and c2, lastpos() is defined as follows:

lastpos(+) = lastpos(c1) U lastpos(c2) (6.8)

lastpos(.) = if (nullable(c2) == true) then
return (lastpos(c1) ∪ lastpos(c2))
else
return lastpos(c2) (6.9)

The difference between the firstpos() and the lastpos() in only for the sub-tree that involves the concatenation operator and for all other nodes, firspos() and lastpos() are same. This function is computed for all interior and leaf nodes and the result is stored in a list.

The summary of all the functions are listed in Table 6.1

Table 6.1 Summary of the functions nullable(), firstpos() and lastpos()

Node *n*	***nullable*(*n*)**	***firstpos*(*n*)**	***lastpos*(*n*)**
Leaf ε	true	∅	∅
Leaf i	false	{i}	{i}
+ or (|) / \\ c_1 c_2	nullable(c_1) or nullable(c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
• / \\ **c1 c2**	nullable(c1) and nullable(c2)	if nullable(c1) then firstpos(c1) ∪ firstpos(c2) else firstpos(c1)	if nullable(c2) then lastpos(c1) ∪ lastpos(c2) else lastpos(c2)
* | **c1**	true	firstpos(c1)	lastpos(c1)

6.2.5 followpos(n)

followpos() is computed only for the leaf nodes that are labeled with the input symbols that constitute the language of the regular expression. followpos(i) is defined as the set of positions that can follow position õ*i*" in the syntax tree. Hence, this is an important function to construct the DFA. To compute followpos(i), the firstpost() and the laspos() of all the nodes are necessary. The algorithm to compute followpos() is given in algorithm 6.1

1. **for** each node *n* in the tree **do**
2. **if** *n* is a concatenation node with left child c_1 and right child c_2 **then**
3. **for** each *i* in $lastpos(c_1)$ **do**
4. $followpos(i) := followpos(i) \cup firstpos(c_2)$
   end do
5. **else if** *n* is a *-node
6. **for** each *i* in $lastpos(n)$ **do**
7. $followpos(i) := followpos(i) \cup firstpos(n)$
   end do

end if

end do

Algorithm 6.1: Followpos() construction

From Algorithm 6.1 it is visible that the followpos() is determined for all the leaf nodes based on their integers assigned to them. In computation of followpos() the Kleene closure operator node and the concatenation operator nodes are only considered. The union operator is not considered since the union operator can choose the appropriate path to validate the string.

Line 1 of the algorithm, considers all the nodes of the syntax tree. In line 2, the node can be analysed based on the concatenation operator and line 5 analyses and constructs followpos()

based on the Kleene closure operator. In line 3, every element present in lastpos() of the left child of the concatenation node is considered and the followpos() value is updated according to the relationship given in line 4 of the algorithm. Similarly, line 6 of the algorithm, considers all the elements present in the lastpos of the * node and determines their followpos() based on the relation given in line 7 of the algorithm. As line 7 involves followpos() of * node, the followpos() of the concatenation operator node is first determined which is then used to identify the followpos() of the Kleene closure operator node.

6.2.6 DFA Construction

The construction algorithm is given in Algorithm 6.2. Initially, the firstpos (root) is considered. The integers representing in this firstpos( ) is considered as one state of the DFA. This is the starting state of the DFA. Line 1 and 2 of the algorithm, indicates, this procedure, where the variable Dstates is used to remember the states of the DFA. Line 3 initializes a while loop which generates new states based on the transition from the initial state. The start state is marked in line 4 to indicate that the state has been considered. Line 5 initiates for loop to define transitions from the start state on all input symbols. The new state will have state numbers which is determined as a union of the state numbers of the followpos() of the state numbers in the initiating state. The procedure is repeated till there are no more new states can be generated.

1. s_0 := firstpos(root) where root is the root of the syntax tree
2. Dstates := {s_0} and is unmarked
3. **while** there is an unmarked state T in Dstates **do**
4. mark T
5. **for** each input symbol a ∈ Σ **do**
6. let U be the set of positions that are in followpos(p)
7. for some position p in T,
   a. such that the symbol at position p is ÷aø
   b. **if** U is not empty and not in Dstates **then**
      add U as an unmarked state to Dstates

      end if
      Dtran[T,a] := U
   end do
end do

Algorithm 6.2 ó DFA Construction

Example 6.1 : Construct a minimized DFA for the regular expression (a|b)*abb

In example 6.1, the regular expression is suffixed with # to form (a|b)*abb#. According to the construction algorithm, given in figures 6.1 (a óc), the syntax tree is constructed and is shown in figure 6.2

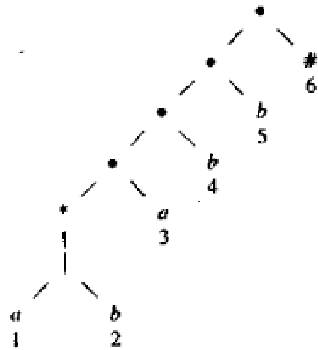

Figure 6.2 Syntax tree construction for (a|b)*abb#

For the syntax tree, the only nullable() node happens to be the * node. Equations 6.1 to 6.3 are used to compute the nullable of all the nodes. All the leaf nodes are not nullable. For the other interior nodes, the interior concatenation node of children * node and node 3, their nullable information is computed as the õ.ö of its children. Hence, the concatenation node will have nullable information set to õFalseö. Similarly, other interior concatenation nodes will also have nullable information set to õFalseö.

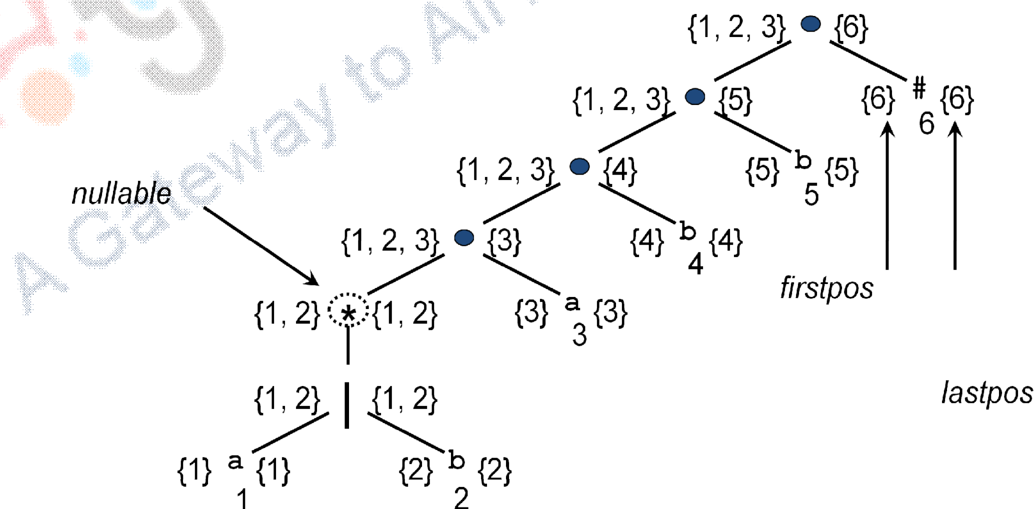

Figure 6.3 Syntax tree with firstpos() and lastpos() marked.

After computing the nullable information, equations 6.4 to 6.9 are used to compute the firstpos() and lastpos() of all the nodes. For the leaf nodes, the numbers indicated will be their firstpos()

and lastpos() and is indicated in figure 6.3. Consider, the union operator node of the children õ1ö and õ2ö. The firstpos(|) and the laspos(|) is given by the union of the firstpos and lastpos of its children. Hence, it is given by {1,2}. The interior node, õ.ö, which is a parent of * and (3), the firstpos(.) is computed as union of the firstpos() of its children since c1 being the * node is nullable. The computation of firstpos() of all the other interior nodes is computed as firstpos() of its left child c1 since, c1 is not nullable for all other interior nodes. On the other hand, the lastpos(.) will be computed based on c2. As the right child of all interior node is not nullable, the lastpos, is computed as just the lastpos (c2) only.

Then using this information, followpos() is computed. At the star node, the followpos(1) and followpos(2) will be the set {1,2} using line 7 of algorithm 6.1. Consider the parent of the * node and we add {3} to the followpos(1) and followpos(2) using line 4 of algorithm 6.1. The followpos() of all the nodes is tabulated in Table 6.2

Table 6.2 – Followpos() of figure 6.3

Node	**Followpos(n)**
1	{1, 2, 3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	

After computing the followpos(), the DFA is constructed by using rootøs firstpos() as the start state. The rootøs firstpos() is {1,2,3} and hence, this set is the start state. In this set, õ1ö and õ3ö corresponds to the input symbol õaö. Hence, to define the edge from this state on õaö, we need to consider the followpos(1) and followpos(3) which is {1,2,3} and {4} respectively. So, we define an edge from {1,2,3} to {1,2,3,4} on the input õaö. Similarly, the state õ2ö corresponds to input õbö. Hence, followpos(2) is to be considered for the edge on õbö from {1,2,3} which is the same state itself. Now, we have a new state {1,2,3,4}. From this state, õ1ö and õ3ö corresponds to õaö and hence, there is a self loop on this state on the input symbol õaö. For õbö, nodes 2, 4 corresponds and hence the union of their followpos() is {1,2,3,5} which is again a new state. The process is repeated and the resultant DFA is shown in figure 6.4. The final state corresponds to the states in which the number corresponding to # is available. In this example, the state number is represented by õ6ö. In the states of the DFA, õ6ö is contained in the set representing {1,2,3,6} and thus it corresponds to the final state of the DFA. Thus we end up with 4 states in the DFA as against 5 states which is the resultant of Thompsonøs construction algorithm.

Pathshala पाठशाला A Gateway to All Post Graduate Courses

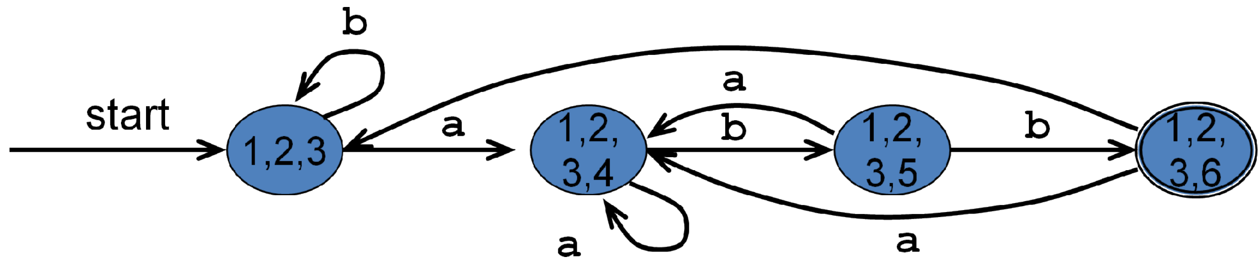

Figure 6.4 Minimized DFA for the expression (a|b)*abb

Summary

In this module, we discussed the construction of minimized DFA which is performed from the regular expression. This algorithm will be proportional asymptotically to the number of operators in the input expression and the number of input symbols present in the regular expression. The next module will discuss the minimized DFA construction from the input DFA.

ePathshala पाठशाला A Gateway to All Post Graduate Courses