Module 7 - Minimized DFA and Lexical Errors

The main objective of this module is to construct a minimized DFA from an already constructed DFA using Thompsonôs construction algorithm.

7.1 Table filling minimization algorithm

The syntax tree procedure which is used to construct a DFA from a regular expression results in a minimized DFA. However, the Thompsonôs construction algorithm discussed in the previous modules results in a DFA which has many redundant states. As we know a DFA is used to perform faster string matching, the DFA needs to have optimum number of states. So, we try to use a minimization algorithm, whose input will be a DFA having a higher number of states and output will be a DFA with optimum number of states. This algorithm is referred as the Table-filling minimization algorithm.

7.1.1. Algorithm

The minimization algorithm takes a DFA as input. This DFA could be the one directly constructed or it could be the one which is constructed using the Thompsonôs construction algorithm. The basics behind this algorithm is to group the states of the DFA based on some input string and identify as many groups as possible in an iterative manner. Finally, the states in a single group indicate that they cannot be distinguished and hence can be reduced to single state. Construct the DFA and then use this procedure to eliminate redundant states.

At the beginning of the procedure, we assume two distinguished groups of states:

1. The group of non-accepting states
2. The group of accepting states.

Then we use the method of partition of equivalent class on input string to partition the existing groups into smaller groups. The main idea is to consider a group of the current partition say A = {p, q, r, é } and an input symbol ñaò and check if ñaò can be used to distinguish between any states in group A. We consider the transitions from every state in A on the input symbol ñaò and if the states reached fall into two or more groups of the current consideration, we split ñAò into a collection of groups so that ñpò and ñqò are in the same group if and only if they go to the same group on input ñaò. We repeat the process until no group, and no input symbol can the group be split any further. The algorithm is discussed in Algorithm 7.1

Algorithm 7.1

Input. A DFA M={S,Σ, ŭ, s_0,F}

Output. A DFA Mô= {Q, Σ, ŭ, s_0, R}, | Q | Ò | S | and | R | Ò| F | accepting the same language as M

MinimizedDFA(M)

{

1. Construct an initial partition Ø of the set of states with two groups: the accepting states *F* and the non-accepting states *S*-*F*. $Ø_0 = \{I_0^1, I_0^2\}$

2. For each group *I* of $\prod_i$,partition *I* into subgroups such that two states *s* and *t* of *I* are in the same subgroup if and only if for all input symbols *a*, states *s* and *t* have transitions on *a* to states in the same group of $\prod_i$; replace *I* in $\prod_{i+1}$ by the set of subgroups formed.

3. If $\prod_{i+1} = \prod_i$,let $\prod_{final} = \prod_{i+1}$ and continue with step (4). Otherwise, repeat step (2) with $\prod_{i+1}$

4. Choose one state in each group of the partition $\prod_{final}$ as the representative for that group which are the representatives and will be the states of the reduced DFA Mô

5. Let *s* and *t* be representative states for *s*ôs and *t*ôs group respectively, and suppose on input *a* there is a transition of *M* from *s* to *t*. Then *M'* has a transition from *s* to *t* on *a*.

6. If Mô has a dead state (a state that is not accepting and that has transitions to itself on all input symbols), then remove it. Also remove states that are not reachable from the start state.

7. The meaning is that string *w distinguishes* state *s* from state *t b*y starting with the DFA M in state s and giving it input w, ends up in an accepting state

   (or)

   Starting in state t and giving it input w, ends up in a non-accepting state.

}

Step 1 of the algorithm, is an initialization routine, where the states are grouped into ñAccepting Statesò and ñNon-accepting Statesò. Step 2 and 3 iterates, and keep on splitting the states into groups such that two states ñsò and ñtò will be in the same group if and only if for all input symbols ñaò, transitions from ñsò and ñtò will be in the same group. If they donôt go to the same state, split this group into further groups. After dividing into groups, the states that are in one group indicate equivalent states. From each group, one representative will be chosen and that will be the state in the minimized DFA. This is stated in Step 4 of the algorithm. After deciding on the representative states, the input DFA is referred and the transition is completed in the output DFA. If any state that belongs to the input is not reachable from the start state of the output DFA, then that state is considered as a dead state and is not included in the minimized DFA. This is summarized in Step 6 to 8 of the algorithm to remove redundant states. The following example explains this algorithm to construct a minimized DFA.

7.1.2 Examples

Example 7.1 Construct a minimized DFA for the following DFA given by the transition diagram of figure 7.1

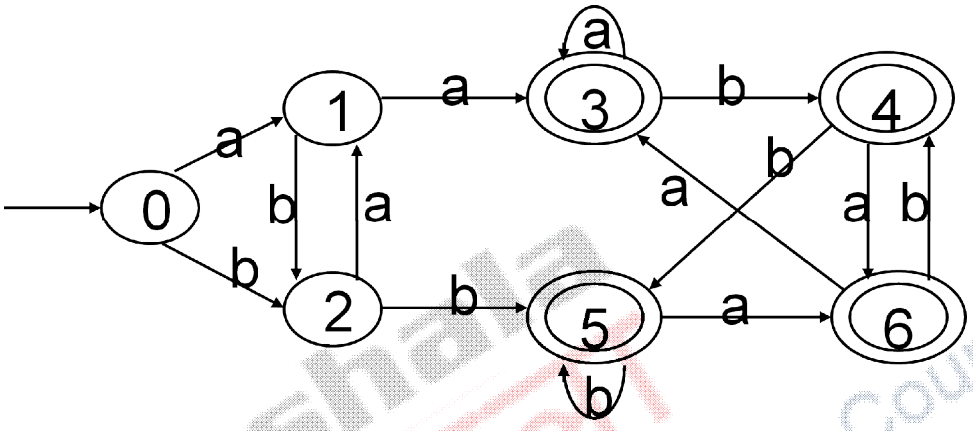

Figure 7.1 DFA to be minimized as per algorithm 7.1

From figure 7.1, in the input DFA there are 3 non-final states and 4 final states. Hence, according to step 1 of the algorithm, the initial grouping is done as

Initialization: $\emptyset_0 = \{\{0,1,2\},\{3,4,5,6\}\}$

Then, each of these groups are considered separately and Step 2 to 3 of the algorithm are applied to form groups. Consider the non-accepting states group in $\emptyset_0$:

- On the input a: move(0,a)={1} ; and move (2,a) = {1}. move({1},a)={3} . On input symbol óaô, states 0 and 2 go to the same group, whereas state 1 goes to a state that is in another group. Hence, we need to split {0,1,2} into two groups, resulting in,

  ï $\emptyset_1$`={{1}, {0,2} }

- On the input b: move (0,b)={2}; move(1,b)={2}; move({2},b)={5}. . On input symbol ób ô, states 0 and 1 go to the same group, whereas state 2 goes to a state that is in another group of $\emptyset_1$. So, we need to split {0,2} into two groups {0} and {2}.
- Thus the non-accepting states of the initial partition results in three groups {0}, {1} and {2}.

Consider the accepting states group in $\emptyset_0$:

- On input a: move({3,4,5,6},a)={3,6}, which is the subset of {3,4,5,6}
- Similarly on input b: move({3,4,5,6},b)={4,5}, which is the subset of {3,4,5,6}

Hence, the groups are {{1}, {0}, {2}, {3,4,5,6}}.

Thus after two iterations, we apply steps 1-3 of algorithm 7.1 and the result is the set $Ø_1$,and get $Ø_2$. So, $Ø_{final} = Ø_1$

Let us consider a representative from each group. For groups 0, 1, 2, the states themselves are the representative and let state 3 represent the group {3,4,5,6}. The construction of the DFA is done by referring to the input DFA. The transitions on input symbols ñaò and ñbò from states 0, 1, 2 are the same as the input DFA. The other transitions leading to any of states 3, 4, 5, 6 would lead to state 3. This is shown in figure 7.2 which is the output DFA.

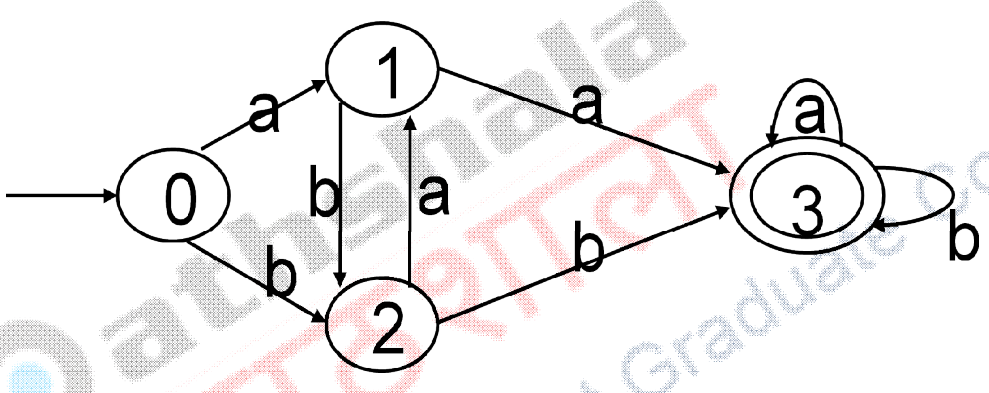

Figure 7.2 Output DFA

Example 7.2 Consider the DFA constructed using Thompson's construction algorithm for the regular expression (a|b)*abb

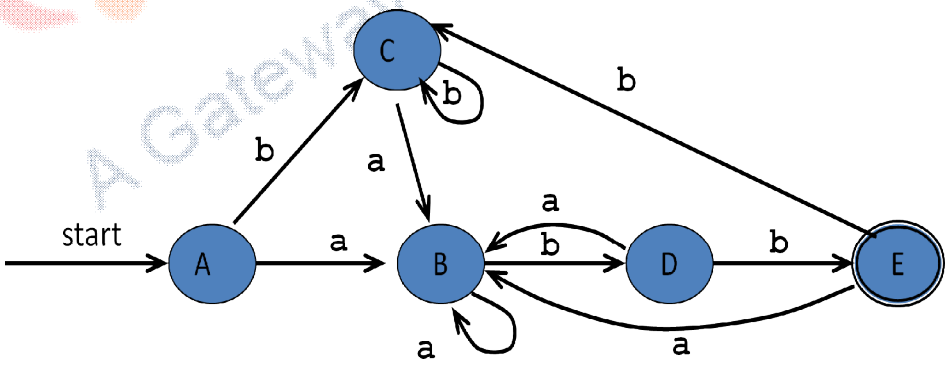

Figure 7.3 Input DFA

From figure 7.3, the initial partition would be {A, B, C, D} and {E} where E is the final state and all other states are non-final states.

Consider the non-final states grouping {A, B, C, D}

- For the input ñbò, move({A, B, C}, b) = {C, D}, while move(D, b) = {E} and hence we need to split.

Split them into two groups {A, B, C} and {D}.

- For the input ñbò, move({A, C}, b) = {C} while move(B, b) = {D}, and hence we split this further into two groups {A,C} and {B}

If we consider the group {A, C}, their moves go to the same state and hence cannot be split further.

Thus this results in the states {A, C}, {B}, {D}, {E} and results in the minimized DFA. This DFA is exactly same as the one which is constructed using syntax tree procedure and is given in Figure 7.4.

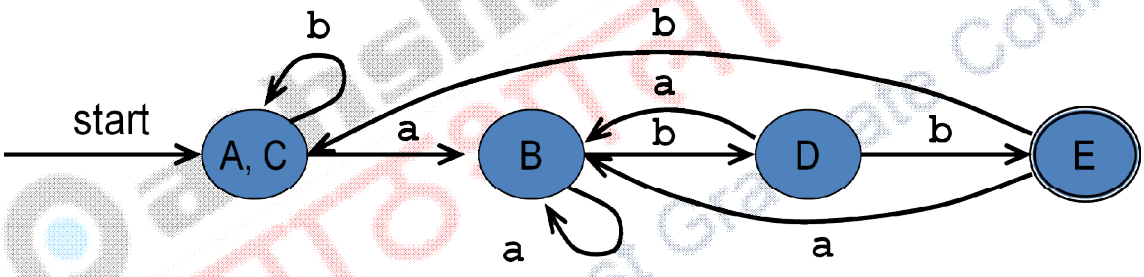

Figure 7.4 Minimized DFA for (a|b)*abb

7.2 Token Recognition

Till now we have discussed the ways to define regular expressions and to recognize patterns using NFA/DFA. We have also discussed to convert regular expression to a DFA as it is helpful in faster string matching. After identifying the input string using the DFA or regular expression, the final state of the automata need to return the lexeme encountered along with a token. The task of recognition of token in a lexical analyzer aids in the following:

- ï Isolate the lexeme for the next token in the input buffer
- ï Produce ~~as~~ an output a pair consisting of the appropriate token and attribute-value, such as <id, pointer to table entry> , using the translation table

Few examples of tokens along with their regular expression and attribute is given in Table 7.1

Table 7.1 Examples of few tokens and their attributes

Regular Expression	Token	Attribute Value
if	**if**	-
id	**id**	Pointer to table entry
<	**relop**	LT

After associating a token and attribute value for every lexeme, the information is stored in the Symbol Table for reference by the other phases of the compiler.

7.3 Lexical Errors

If the input contains errors then the Automata or the regular expression may not be in a position to match and hence the input cannot be recorded in the symbol table. In order to proceed with the compilation the compiler normally recovers from errors. The simplest recovery strategy is the ñpanic modeò. The following are the few panic modes of error recovery which the lexical analyzer adopts to recover from errors:

- Deleting an extraneous character ï Keep on deleting extra characters till a possible match is found. Example: If the token encountered is ñifxò, where ñxò is an extraneous character, then this character is deleted.
- Inserting a missing character ï Insert any one or two missing characters to match a given regular expression. Example: If the token encountered is ñflatò, a character ñoò if inserted will get a correct token ñfloatò.
- Replacing an incorrect character by a correct character ï In the event where the given input does not match with a pattern, replacing with one character might get a correct pattern. Example: if the token is ñingò, replacing the ógô with ótô will get a correct token ñintò.
- Transposing two adjacent characters ï Swapping of two adjacent characters to get a valid token. Example: If the token encountered is ñfiò then swapping two characters will result in ñifò.
- Pre-scanning ï Going through multiple passes to identify and recover from errors.

7.4 Input Buffering

In order to recognize the lexemes to associate a token and attribute to it, some ways need to be identified to read the source programs. This task is given importance since the compiler has to look ahead one or more characters, beyond the next lexeme. For example, if the string in the input is ñ<=ò, the lexer can recognize it as ñ<ò or ñ<=ò corresponding to either the relational operator ñless thanò or ñless than or equal toò. The lexer handles this problem by always

choosing the longest matching pattern and then associates a token and attribute for the same. In order to handle this efficiently, two-buffer input scheme to look ahead on the input and identify tokens are proposed.

7.4.1 Buffer pairs

In order to handle large amount of data that need to be processed efficiently specialized buffering techniques have been developed to help avoid the overhead to process a single character. The scheme involves two buffers that are alternately reloaded and is given in figure 7.5

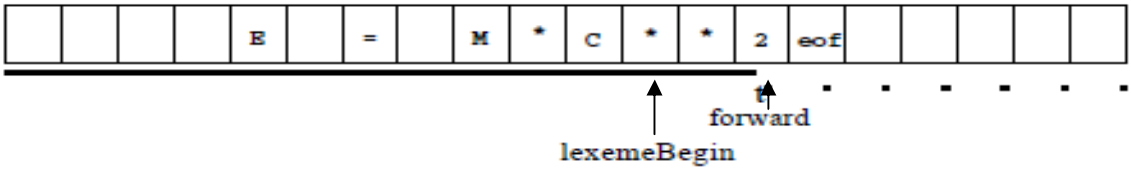

Figure 7.5 Using a pair of input buffers

Each buffer is of the same size N and is the size of a disk block. Using one system read command; N characters are read into a buffer. If the number of characters is less than N, then a special character ñeofò is introduced at the end of the buffer. The buffering scheme uses two pointers to identify the lexeme.

- ñlexemeBeginò pointer marks the beginning of the current lexeme
- ñforwardò pointer keeps moving forward to identify the longest matching pattern.

The forward pointer keeps moving and once it detects the next pattern based on the regular expression or automata, then the characters between, the ñlexemeBeginò pointer and the previous character to the ñforwardò pointer is identified as the lexeme. Then the lexemeBegin pointer is reset to the current position of the forward pointer and the forward pointer continues to move to identify the next pattern.

Once, one half of the buffer is over, that buffer is reloaded with the next remaining input characters. Hence, problems could arise if the lexemeBegin pointer is in the first half of the buffer and the forward pointer is in the other buffer half. In such situation the lexeme is lost and to avoid this we go for Sentinel based input buffering scheme.

7.4.2 Sentinels

In order to avoid losing the ñlexemeBeginò information, the special symbol ñeofò is appended at the end of each buffer half. Thus, if the input has to be loaded into one buffer half and if the ñlexemeBeginò pointer is in that half, the prefix string is saved into a temporary variable and processed. Thus the eof acts as end marker of each buffer half.

Summary

This module discussed the construction of a minimized DFA from the input DFA. The lexical errors that could be encountered and ways of recovering also discussed. Input buffering schemes to process the input and identify the lexeme were also discussed. The next module discusses the lexical analyser generator using a tool.