

The Language for Specifying Lexical Analyzer

We shall now study how to build a lexical analyzer from a specification of tokens in the form of a list of regular expressions. The discussion centers around the design of an existing tool called **LEX**, for automatically generating lexical analyzer program.

A LEX source program is a specification of a lexical analyzer, consisting of a set of regular expressions together with an action for each regular expression. The action is a piece of code which is to be executed whenever a token specified by the corresponding regular expression is recognized. The output of LEX is a lexical analyzer program constructed from the LEX source specification.

Unlike most programming languages, a source program for LEX does not supply all the details of the intended computation. Rather, LEX itself supplies with its output a program that simulates a finite automaton. This program takes a transition table as data. The transition table is that portion of LEX's output that stems directly from LEX's input.

The situation is shown in Fig.25. Where the lexical analyzer L is the transition table plus the program to simulate an arbitrary finite automaton expressed as a transition table. Only L is to be included in the compiler being built.

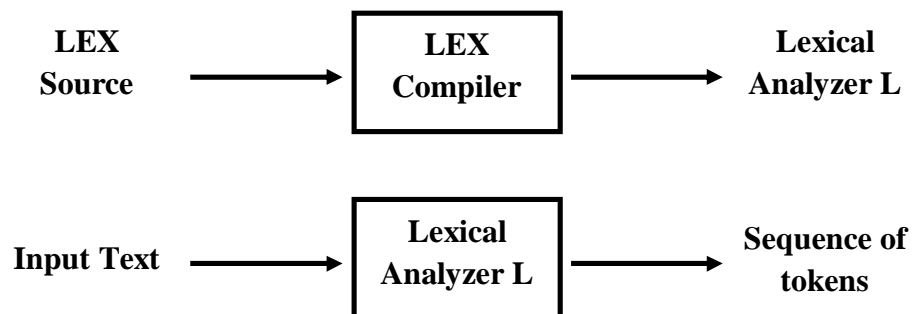


Fig.25: the Role of LEX

A LEX source program consists of two parts, a sequence of auxiliary definition followed by a sequence of translation rules.

Auxiliary Definitions

The auxiliary definitions are statements of the form:

$$D_1=R_1$$
$$D_2=R_2$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$D_n=R_n$$

Where each D_i is a distinct name, and each R_i is a regular expression whose symbol are chosen from $\Sigma \cup \{D_1, D_2, D_{i-1}\}$, i.e., characters or previously defined names. The D_i 's are shorthand names for regular expressions. Σ is our input symbol alphabet.

Example: We can define the class of identifiers for a typical programming language with the sequence of auxiliary definitions.

$$\text{Letter} = A \mid B \mid \dots \mid Z$$
$$\text{Digit} = 0 \mid 1 \mid \dots \mid 9$$
$$\text{Identifier} = \text{Letter} (\text{Letter} \mid \text{Digit})^*$$

Translation Rules

The translation rules of a LEX program are statements of the form:-

$$P_1 \quad \{A_1\}$$
$$P_2 \quad \{A_2\}$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$P_m \quad \{A_m\}$$

Where each P_i is a regular expression called a pattern, over the alphabet consisting of Σ and the auxiliary definition names. The patterns describe the form of the tokens. Each A_i is a program fragment describing what action the lexical analyzer should take when token P_i is found. The A_i 's are written in a conventional programming language, rather than any particular language, we use pseudo language. To create the lexical analyzer L, each of the A_i 's must be compiled into machine code.

The lexical analyzer L created by LEX behaves in the following manner: **L read its input, one character at a time, until it has found the longest prefix of the input which matches one of the regular expressions, P_i . Once L has found that prefix, L removes it from the input and places it in a buffer called TOKEN. (Actually, TOKEN may be a pair of pointers to the beginning and end of the matched string in the input buffer itself). L then executes the action A_i .**

It is possible, that **none of the regular expressions** denoting the tokens matches any prefix of the input. In that case, an **error has occurred**, and L transfers control to some error handling routine. It is also possible that two or more patterns match the same longest prefix of the remaining input. If that is the case, L will break the tie in favor of that token which came first in the list of translation rules.

Example: Let us consider the collection of tokens defined in Fig.7, LEX program is shown in Fig.26

AUXILIARY DEFINITION

Letter= A | B | ... | Z

Digit= 0 | 1 | ... | 9

TRANSLATION RULES

BEGIN	{return 1}
END	{return 2}
IF	{return 3}
THEN	{return 4}
ELSE	{return 5}
letter(letter digit)*	{LEX VAL:= INSTALL(); return 6}
digit ⁺	{LEX VAL:= INSTALL(); return 7}
<	{LEX VAL := 1; return 8}
<=	{LEX VAL := 2; return 8}
=	{LEX VAL := 3; return 8}

< >	{LEX VAL := 4; return 8}
>	{LEX VAL := 5; return 8}
>=	{LEX VAL := 6; return 8}

Fig.26: LEX Program.

Suppose the lexical analyzer resulting from the above rules is given input BEGIN followed by blank. Both the first and sixth pattern matches BEGIN, and no pattern matches a longer string. Since the pattern for keyword BEGIN precedes the pattern for identifier in the above list, the conflict is resolved in favor of the keyword.

For another example, suppose <= are the first two characters read. While pattern < matches the first character, it is not the longest pattern matching a prefix of the input. Thus LEX's strategy that the longest prefix matching a pattern is selected makes it easy to resolve the conflict between < and <=, by choosing <= as the next tokens.

Implementing the Lexical Analyzer

The LEX can build from its input a lexical analyzer that behaves roughly like a finite automaton. The idea is to construct a NFA N_i for each tokens pattern P_i in the translation rules, then links these NFA's together with a new start states as shown in Fig.27. Next we convert this NFA to a DFA.

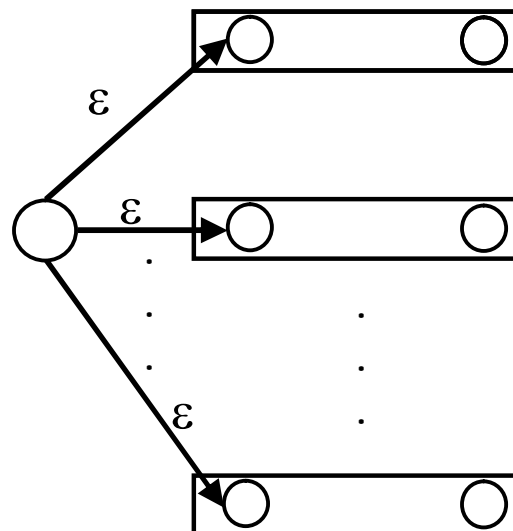


Fig.27: NFA recognize several tokens simultaneously

There are several nuance in this procedure of which the reader should be aware. **First**, there are in the combined NFA several different "accepting states". That is, the accepting state of each N_i indicates that its own token, P_i has been found. When we convert to a DFA, the subsets we construct may include several different final states. Moreover the final states lose some of their significance, since we are looking for the longest prefix of the input which matches some patterns. After reaching a final state, the lexical analyzer must continue to simulate the DFA until it reaches a state with no next state for the current input symbol.

Upon reaching termination, it is necessary to review the states of the DFA which we have entered while processing the input. Each such state represents a subset of the NFA's states, and we look for the last DFA state which includes a final state for one of the pattern-recognizing NFA's N_i . That final state indicates which token we have found. If none of the states which the DFA has entered includes any final state of the NFA, then we have an error condition. If the last DFA state to include a final NFA state in fact includes more than one final state, then the final state for the pattern listed first has priority.

Example: Suppose we have the following LEX program.

AUXILIARY DEFINITION

(none)

TRANSLATION RULES

a	{ }	/* actions are omitted here*/
abb	{ }	
a*b ⁺	{ }	

The three tokens above are recognized by the simple automata of Fig.28. We may convert the NFA's of Fig.28 into one NFA as described earlier. The result is shown in Fig.29. Then this NFA may be converted to a DFA using the algorithm and the transition table is shown in Fig.30, where the states of the DFA have been named by lists of the states of the NFA.

The last column in Fig.30 indicates the token which will be recognized if that state is the last state entered that recognizes any token at all. In all cases but the last line, state 68, the token recognized is the only token whose final state is included among the NFA states forming the DFA state. For example, among NFA states 2, 4, and 7,

only 2 is final, and it is the final state of the automaton for regular expression 'a' in Fig.28. Thus, DFA state 247 recognizes token 'a'. In the case of DFA state 68, both 6 and 8 are final states of their respective nondeterministic automata. Since the translation rules of our LEX program mention **abb** before **a*b⁺**, NFA state 6 has priority, and we announce that abb has been found in DFA state 68.

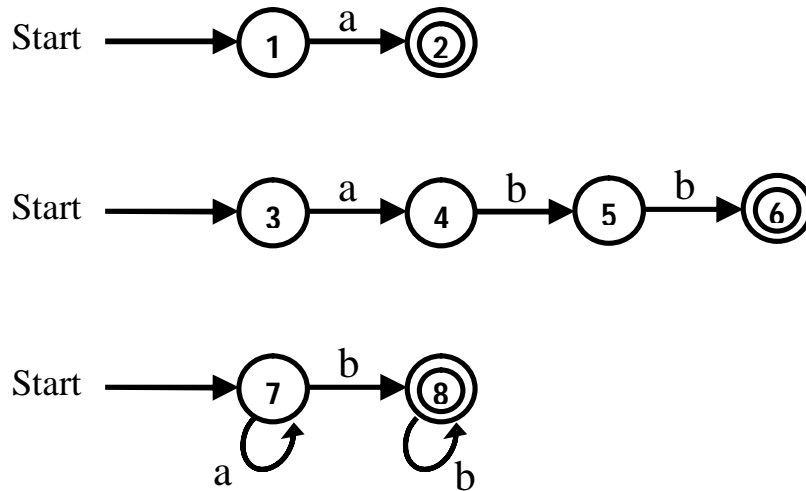


Fig.28: Three tokens are recognized by the simple automata

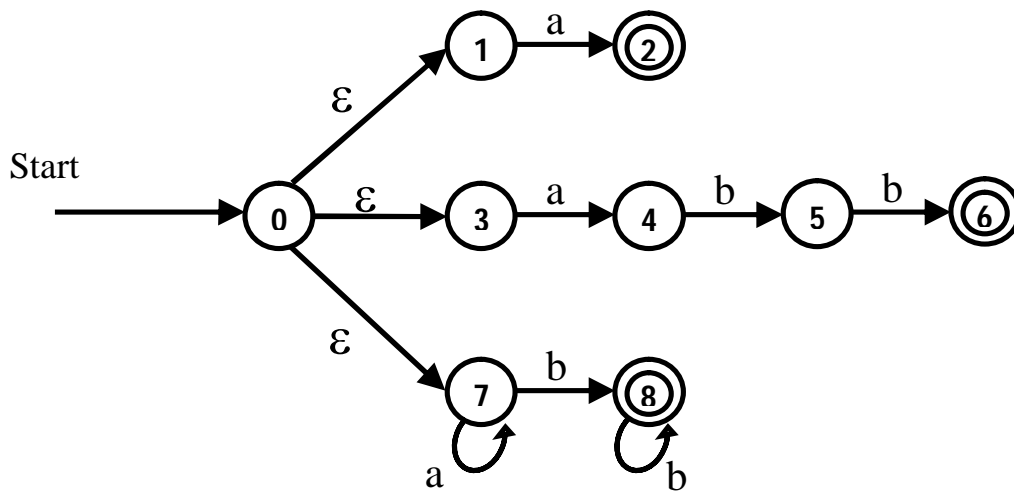


Fig.29: NFA recognizing three different tokens

<i>State</i>	<i>a</i>	<i>b</i>	<i>Token found</i>
0137	247	8	none
247	7	58	a
8	---	8	a^*b^+
7	7	8	none
58	---	68	a^*b^+
68	---	8	abb

Fig.30: Transition Table for DFA.

Suppose that the first input characters are **aba**. The DFA of Fig.30 starts off in state 0137. On input 'a' it goes to state 247. Then on input 'b' it progresses to state 58, and on input 'a' it has no next state. We have thus reached termination, progressing through the DFA states 0137, then 247, then 58. The last of these includes the final NFA state 8 from Fig.28. Thus the action for state 58 of the DFA is to show that the token a^*b^+ has been recognized and to select **ab**, the prefix of the input that led to state 58, as TOKEN.

If the DFA state 58, that is the last state entered before termination not include a final state of some NFA, then we would consider the DFA state previously entered, that is state 247 and recognize the token 'a', which state 247 call for. The prefix 'a' would in that case be TOKEN.

In Fig.31 we see NFA's for the patterns of Fig.26. We use here a simple NFA for identifiers and constants.

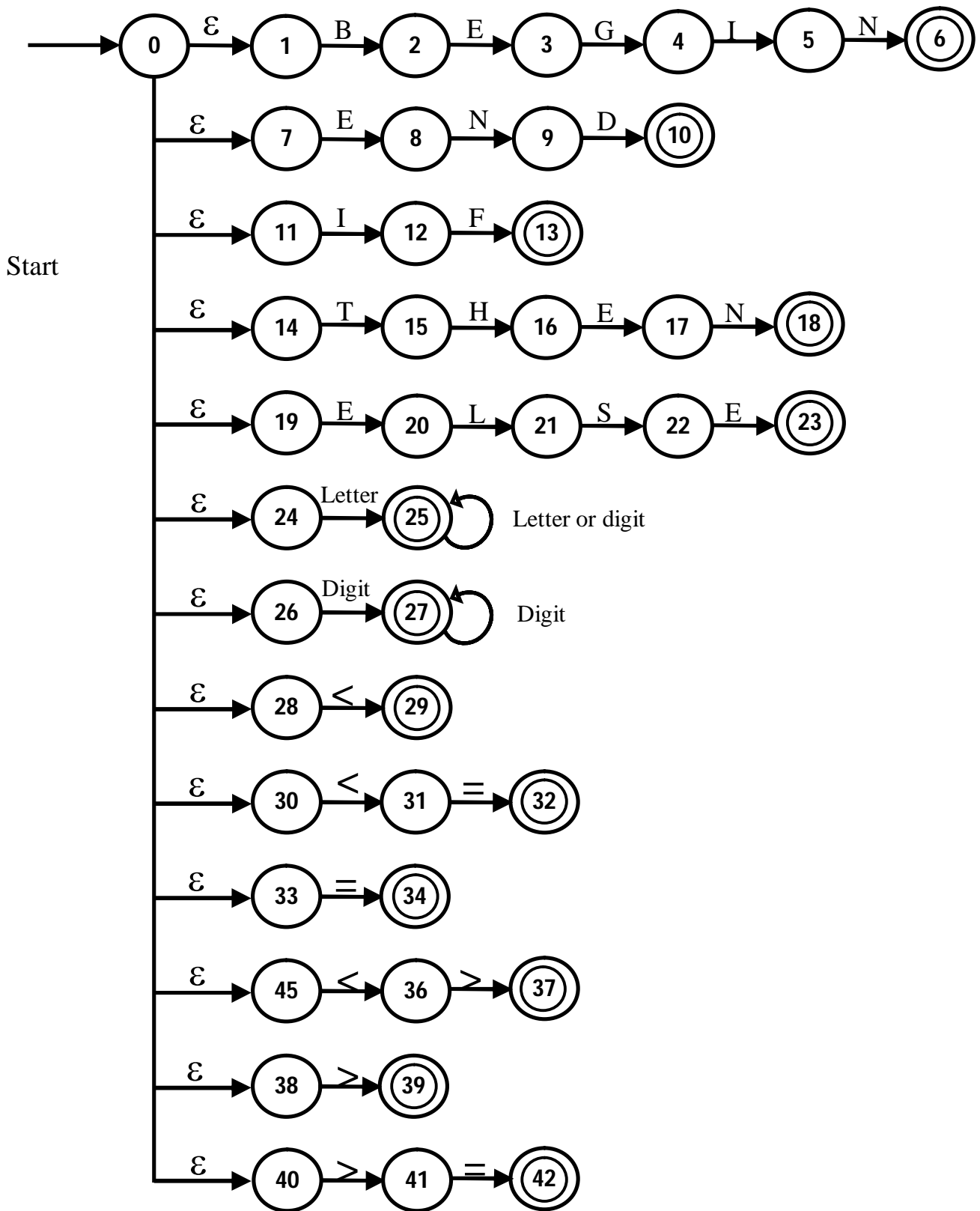


Fig.31: Combined NFA for Tokens.

The DFA constructed from Fig.31 is shown in Fig.32. Each state of the deterministic automata is a subset of the states of the nondeterministic automaton. State S_0 is the set of states $\{0, 1, 7, 11, 14, 19, 24, 26, 28, 30, 33, 35, 38, 40\}$

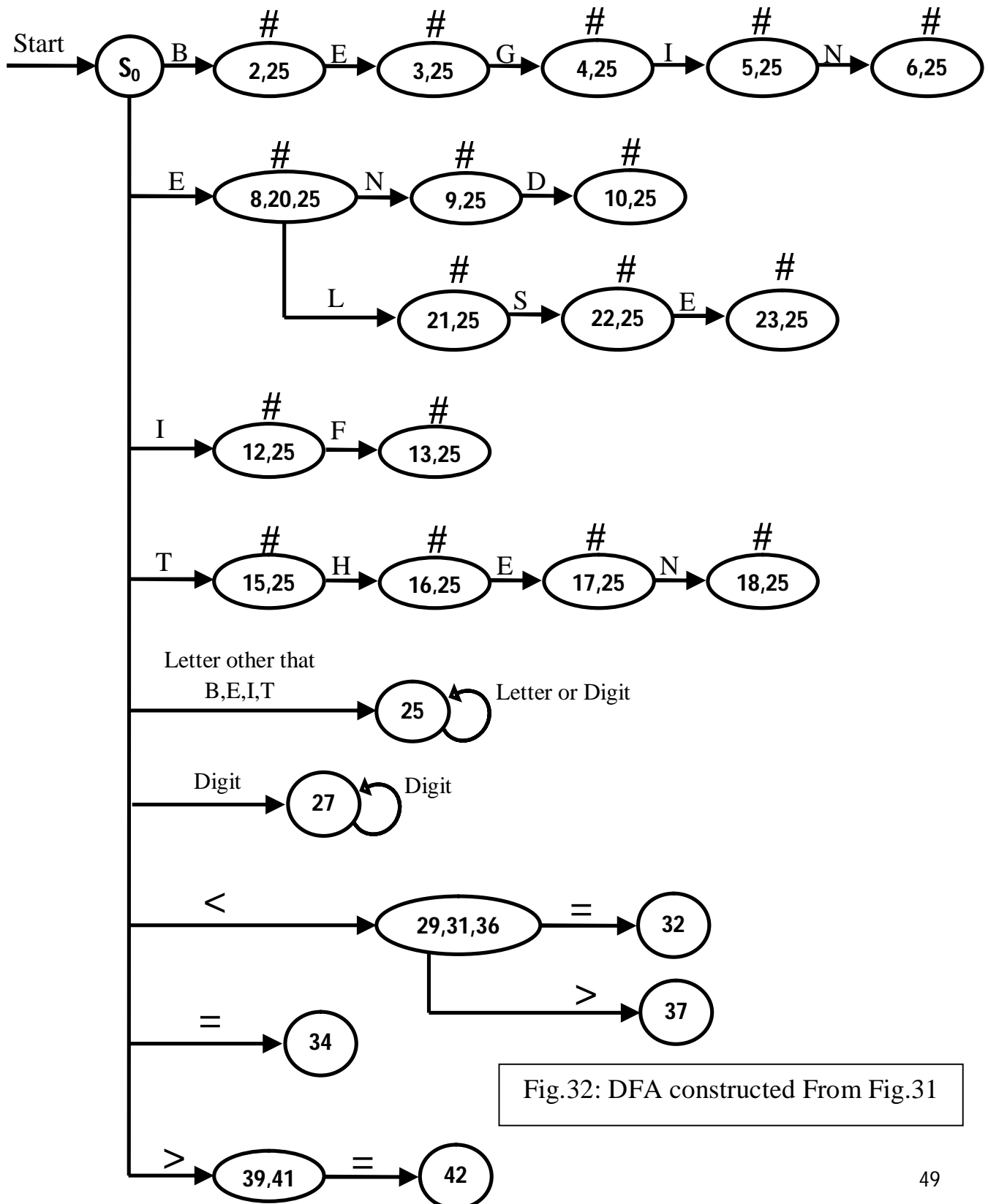


Fig.32: DFA constructed From Fig.31

The states marked # have transition to state {25} on all letters and digit for which no transition is shown.

On input such as 'THE' followed by a non letter or digit, the DFA break in state {17, 25}, which contain accepting state 25 of the nondeterministic automaton. That state signals that an identifier has been found and the appropriate action is taken.

On input '<A', the lexical analyzer goes to state {29, 31, 36} on '<', from which no transition on 'A' is possible. On the three state of NFA, only state 29 is accepting and it indicates that the token '<' has been found, and take the specified action.

If we want to minimize the DFA, we must use one of the methods for minimization, and after that we see that the result DFA is the same.