Compiler Design (CSE)

LEXICAL ANALYSIS-II

Recognition of tokens

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

Figure 3.10: A grammar for branching statements

Terminals of the grammar:

if, then, else, relop, id, number

The terminals of the grammar, which are **if**, **then**, **else**, **relop**, **id**, and **number**, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions, as in Fig. 3.11. s

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits \ (\texttt{.}\ digits)?\ (\ \texttt{E}\ \texttt{[+-]}?\ digits\ )? \\
letter & \rightarrow & \texttt{[A-Za-z]} \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^{*} \\
if & \rightarrow & \texttt{if} \\
then & \rightarrow & \texttt{then} \\
else & \rightarrow & \texttt{else} \\
relop & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

Figure 3.11: Patterns for tokens of Example 3.8

Token recognition for white space removal:

$$ws \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^{+}$$

Transition diagram

- Patterns converted into Transition diagram.
- Regular Expressions to Transition diagram.
- Set of states (circles)
- Each edge labelled by a symbol or set of symbols.
- Accepting or final states are present.

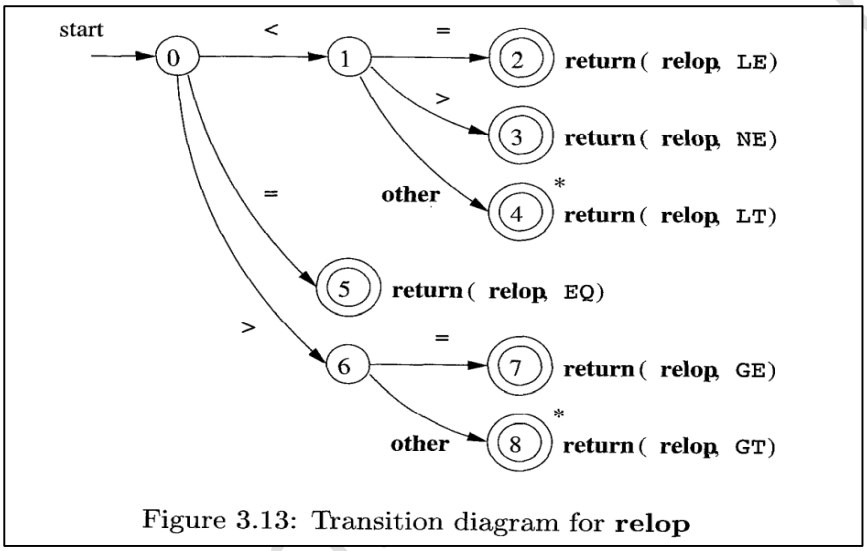


Figure 3.13: Transition diagram for **relop**

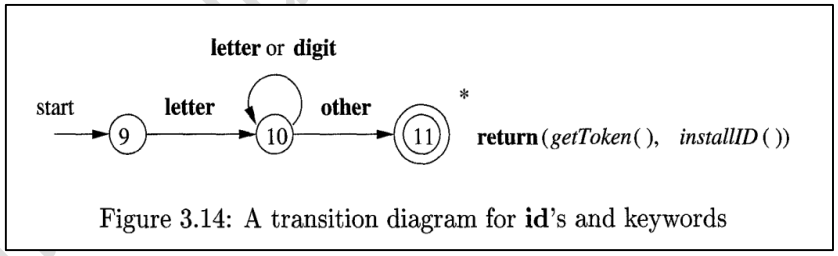


Figure 3.14: A transition diagram for **id**'s and keywords

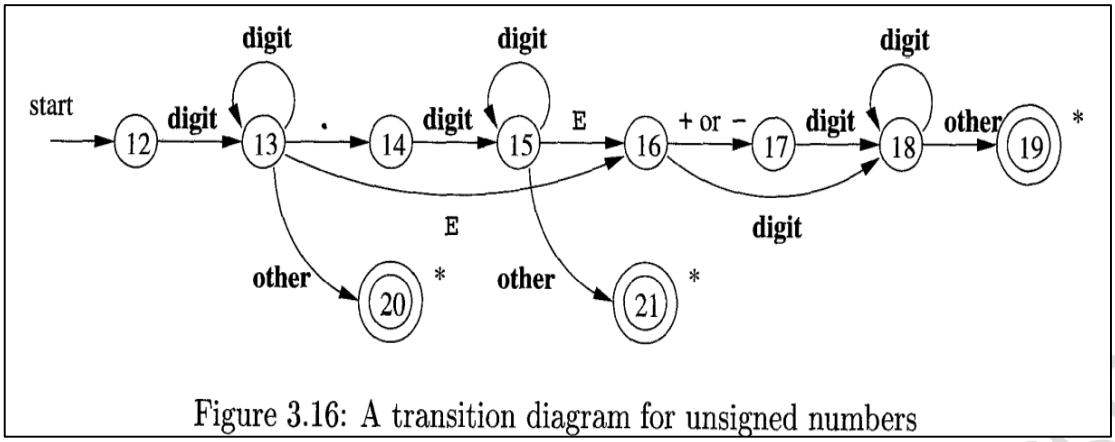


Figure 3.16: A transition diagram for unsigned numbers

Direct Conversion from RE to DFA

Algorithm:- Conversion from RE to DFA (Direct Method)

Input:- A regular expression r

Output:- A DFA that recognizes L(r)

Method:

1. Construct a syntax tree T from augmented regular expression (r)#.
2. Compute nullable, firstpos, lastpos and followpos for T.
3. Construct Dstates (the set of states of DFA D) and Dtran (transition function for D.

Note:

- Dstates are sets of positions in T.
- Initially, each state is "unmarked".
- State becomes "marked" just before we consider its out-transitions
- start is firstpos (root in T).
- Accepting state containing the position for endmark #.

Important definitions / terminologies :

Important states of NFA :-

A state of an NFA is important if it has a non-ϵ out transition.

In subset construction algorithm, only the important states in a set T are used when it computes ϵ-closure (move(T, a)) i.e, the set of states reachable from T on input a.

- Each important state correspond to a particular operand in the RE.
- The constructed NFA has only 1 accepting state, but it is not important as, it's having no out-transition.

Augmented Regular expression :-

concatenate a unique right endmarker # to r, we give a transition on # for accepting state.

So, NFA is now having important states.

Syntax tree :-

Regular expression is represented as a syntax tree. (Post-fix)

- Leaves corresponds to operands.
- Interior nodes → operators.
- Interior nodes called →
  - (i) cat-node ← (concatenation (·) operator)
  - (ii) or-node ← (Union (|))
  - (iii) star-node ← (star (*))
- Cat-nodes represented by hollow circles.
- Leaves are represented by either ϵ or alphabet.
- Each leaf is numbered by integers (except ϵ)
- These position values correspond to important states of NFA.

Functions computed from the syntax tree:

1. Nullable (n) → iff the subexpression represented by n has ϵ in its language.
   i.e, the subexpression can be made null or empty.

2. Firstpos(n) → (Set of positions in the subtree rooted n) that gives the set of first positions that can match first symbol of a string generated by the sub expression.

3. Lastpos(n) → (Set of positions in the subtree rooted n) that gives the set of last positions that can match last symbol of a string generated by the sub expression.

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 $(r)\#$.

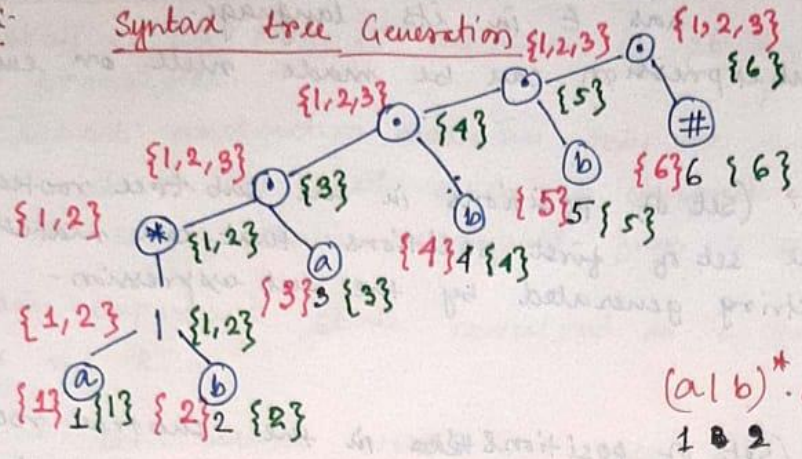
1. *nullable*(n) is true for a syntax-tree node n if and only if the subexpression represented by n has ϵ in its language. That is, the subexpression can be "made null" or the empty string, even though there may be other strings it can represent as well.

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.

Example:- $(a|b)^*a.b.b\#$ (Augmented RE)

Step 1:- Syntax tree Generation



$(a|b)^*.a.b.b.\#$

1 2 3 4 5 6

here (*) node is nullable.

First pos

Follow pos

Last pos.

Followpos Table.

Only for (*) and cat-node (·).

for (*) node :-

for (each i in lp(n))

~~fp(i) = fp(i) ∪~~

followpos (i) = followpos(i) ∪ fp(n);

	Node	Follow pos
a	1	{1, 2, 3}
b	2	{1, 2, 3}
a	3	{4}
b	4	{5}
b	5	{6}
#	6	{ϕ}

For cat-node (·) :-

if (n == $c_1 \cdot c_2$)

for (each i in lp(c_1))

followpos (i) = followpos(i) ∪ fp(c_2)

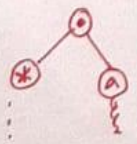
(i)

(*) → i in lp(n) = {1, 2}

follow (1) = {1} ∪ fp(n) = {1} ∪ {1, 2} = {1, 2}

follow (2) = {1, 2}

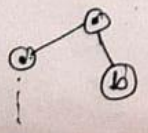
(ii)



→ i in lp(c_1) = {1, 2}

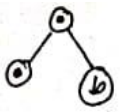
follow (1) = {1, 2} ∪ {3} = {1, 2, 3}

follow (2) = {1, 2} ∪ {3} = {1, 2, 3}

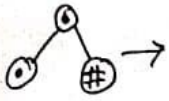


→ i in lp(c_1) = {3}

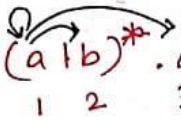
follow (3) = {4}



$LP(4) = \{4\}$
$follow(4) = \{5\}$



$LP(5) = \{5\}$
$follow(5) = \{6\}$.



$(a|b)^* . a . b . b . \#$
1 2 3 4 5 6

$a \rightarrow \{1,2,3\}$
$b \rightarrow \{1,2,3\}$
$a \rightarrow \{4\}$
$b \rightarrow \{5\}$
$b \rightarrow \{6\}$

Now converting RE to DFA (directly).

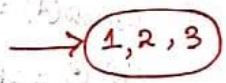
Rule:-

(i) Initial state = fp(root)

(ii) final state = node having position no of #.

(iii) make new state with the help of followpos and node numbers for each symbol transition.

For ex:



1,2,3 initial state

Now, for generating next state,

first state combines the states 1,2,3.

so, for node 1, 2, 3 find the follow positions and make union individually for each symbol (a,b)

Follow(1) = {1,2,3}, Follow(2) = {1,2,3}, Follow(3) = {4}

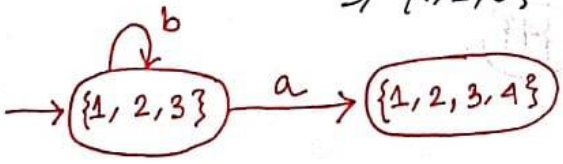
Now, i/p: "a"

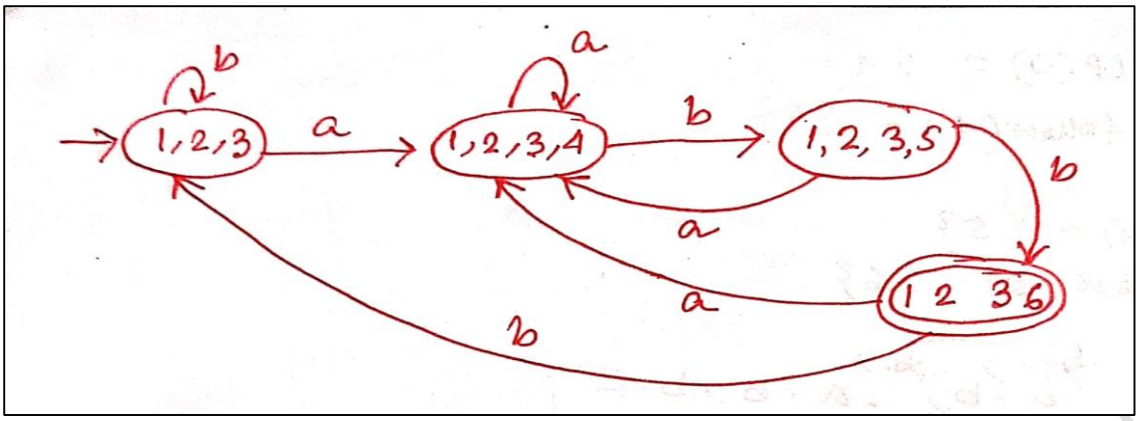
$\delta(\{1,2,3\}, a) \rightarrow$ Follow(1) ∪ Follow(3) [pos 2 = b]

$\Rightarrow \{1,2,3,4\}$.

$\delta(\{1,2,3\}, b) \rightarrow$ Follow(2) [only one symbol pos for b i.e, 2]

$\Rightarrow \{1,2,3\}$.

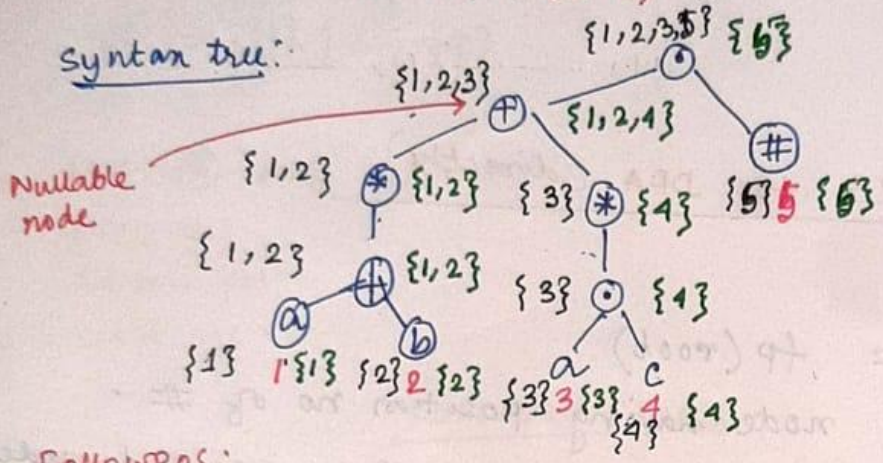




Example-2.

$(a+b)^* + (a \cdot c)^* \#$

Syntax tree:



Followpos:

	Node	Followpos
a	1	{1, 2, 5}
b	2	{1, 2, 5}
a	3	{3, 4}
c	4	{3, 5}
#	5	ϕ

$((a+b)^* + (a \cdot c)^*) \cdot \#$
(1 2 3 4 5)

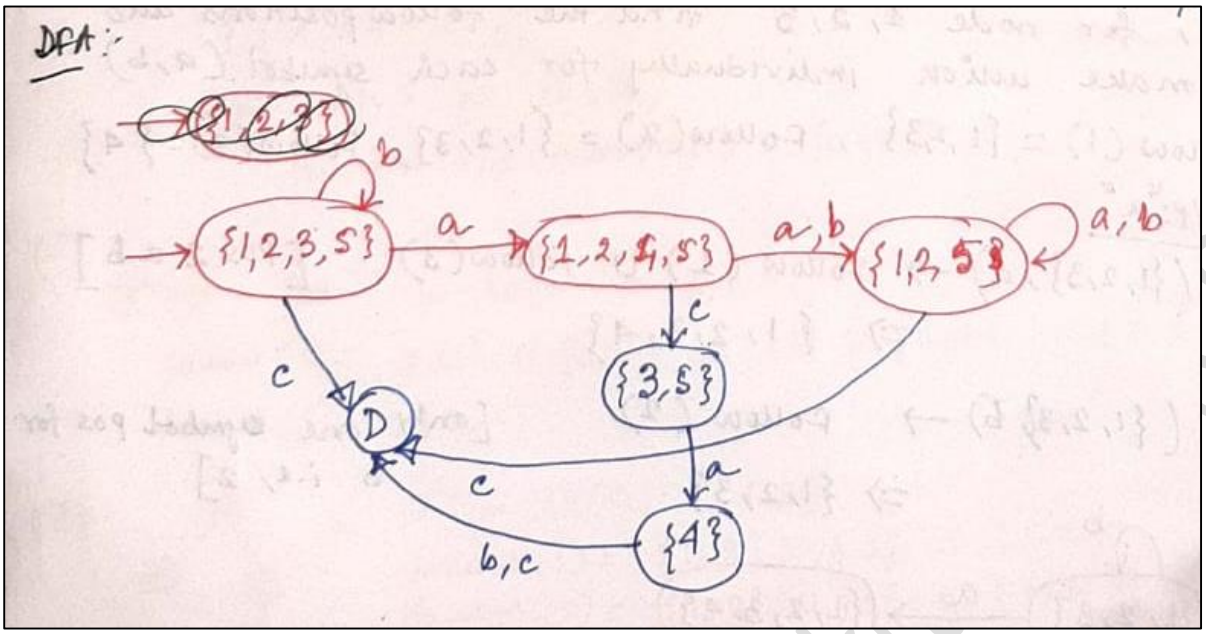
(i) (*) nodes {1,2}

Follow(1) = {1,2}
Follow(2) = {1,2}

(ii) (·) → (3)
Follow(3) = {4}

(*) LP = {4}
Follow(4) = {3}

(iii) (·) = LP = {1,2,4}
Follow(1) = {1,2,5}
Follow(2) = {1,2,5}
Follow(4) = {3,5}



Q. Convert the Regular expression to corresponding DFA (Direct method)

1. **$ab^*a(aba|ba)a^*$**
2. **$a^*b^*a(a|b)^*b^*a$**
3. **$(a|b)^*abb$**

THE LEXICAL-ANALYZER GENERATOR LEX

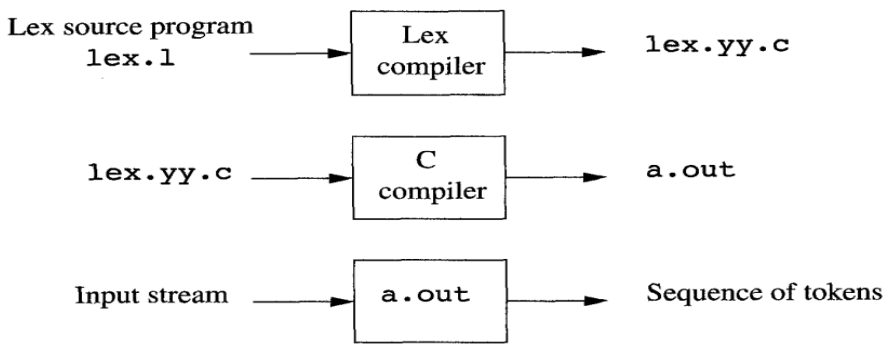


Figure 3.22: Creating a lexical analyzer with Lex

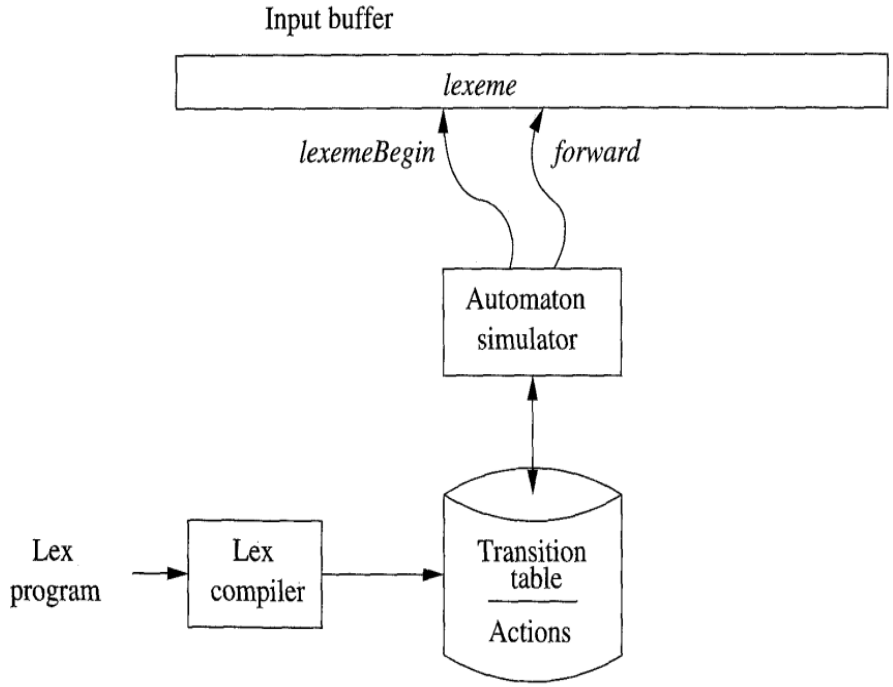


Figure 3.49: A Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator

Errors and their recovery in Lexical Analysis:

- **Errors detected in lexical analysis:**

  1) Numeric literals that are too long

  2) Long identifiers

  3) Ill-formed numeric literals

  4) Input characters that are not in the source language.

- **Error Recovery:**

  1) **Delete:** Unknown characters are deleted (Panic mode Recovery)

  Ex: “charr” corrected as “char”

  2) **Insert :** An extra or missing character is inserted to form a meaningful token. Ex: “Cha” corrected as “char”

  3) **Transpose:** Based on certain rules we can transpose two characters

  Ex: “whiel” can be corrected to “while”

  4) **Replace:** Based on replacing one character by another

Ex: “chrr” can be corrected as “char”