Lex Tool : lexical Analyzer generator.

* Lex is a language for specifying lexical analyzers
* Lex generates programs that can be used in simple lexical analysis of text.
* The input files contain regular expressions for recognizing tokens to be searched for and actions written in C to be executed when expressions are found

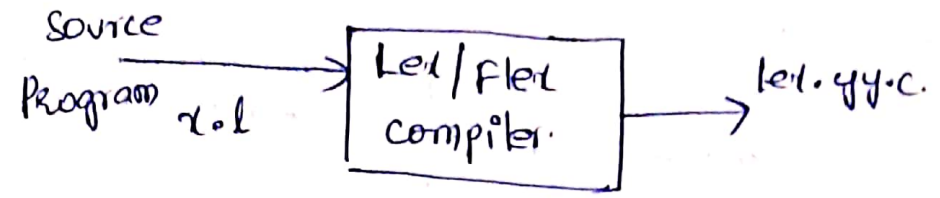
* Lex converts regular expression into table-driven DFA. This deterministic finite automation generated by Lex performs the recognition of the regular expressions.

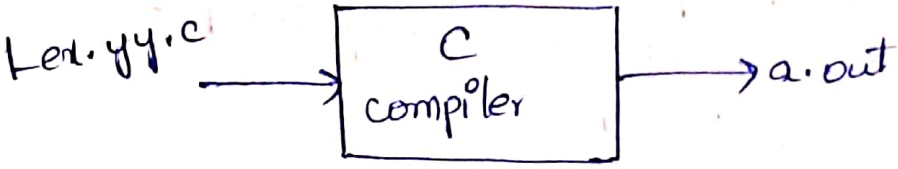
The general form of a lex source file is.

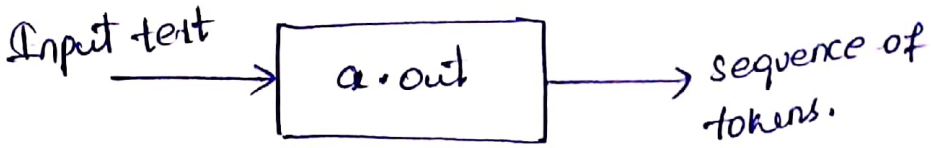
```
Declarations
%%
Regular expressions { actions }
%%
Subroutines.
```

* Lex program contains three sections - declaration, regular expressions, and subroutines.
where the declaration and the user subroutines are often omitted. The second section that is, the regular expressions part is compulsory.
* The declaration part contains 'C' declarations and lex declarations. 'C' declarations are embedded between %{ and %}. Lex declarations contain token definitions.
* when the given input matches with this pattern, action to be performed is described against the regular expression.
* Lex program is stored with an extension ".l". Lex converts the user's expressions and actions into host general-purpose language. The generated program is named yylex() in the lex.yy.c file. This lex.yy.c is the lexer in C.

Assume that lex specification is prepared in file x.l.
Run this input file x.l with lex, that gives lex.yy.c as output
Run lex.yy.c under the C compiler that gives a.out as output.







Regular expressions in Lex use the following operators.

a - the character a
"a" - an "a", even if a is an operator
\a - an "a", even if a is an operator.
[ab] - the character a or b
[a-c] - the character a, b, or c
[^a] - any character but a
. - any character but newline.
^a - an a at the beginning of a line.
<a>b - ab when lex is in start condition a
a$ - an a at the end of a line.
a? - an optional a
a* - 0, 1, 2, ... instances of a
a+ - 1, 2, 3, ... instances of a
a|b - an a or b ab
{aa} - the translation of aa from the definition section
a{m,n} - m through n occurrences of a

Example

Write a lexical analyzer for input

```
Void main( )
{
   int a=10;
}.
```

Lex code:-

```
%{
   #Include<stdio.h>
%}
letter  [_a-z A-Z]
digit   [0-9]
id      {letter}({letter}|{digit})*
%%
{digit}*        printf("%d--- Number\n", yytext);
"int"|"void"|"main"|  Printf("%s--- keyword\n", yytext);
{id}            printf("%s --- Identifier\n", yytext);
"="             Printf("%c ---- assignment operator\n", *yytext);
[)}({]          Printf("%c --- Braces\n", yytext);
","|";"         Printf("%c"--- punctuation symbol\n", yytext);
%%
int main(void)
{ yylex( );
}
int yywrap( )
{ return 1;
}
```

Output

```
void -- keyword
main -- Identifier
(    --- Braces
)    --- Braces
{    --- Braces
int  -- keyword
a    -- Identifier
=    --- assignment operator
10   -- integer
}    --- Braces
;    --- Punctuation symbol
```

Design of a lexical Analyzer Generator

Lexical analyzer can either be generated by NFA or by DFA. DFA is preferable. In the implementation of LEX.

1. Structure of Generated Analyzer:

The architecture of a lexical analyzer generated by LEX is shown below.

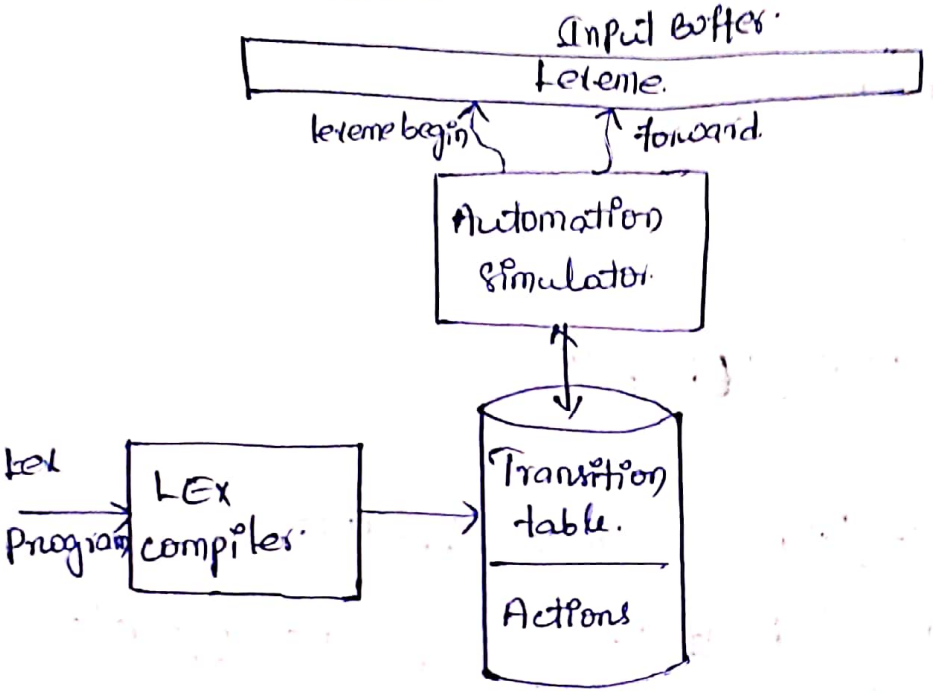


Fig: A lex program is turned into a transition table and actions, which are used by a finite-automation simulator.

* The program that serves as the lexical analyzer includes a fixed program that simulates an automation:

Lexical Analyzer consists of components that are created from the LEX program by LEX itself. These components are

1. A transition table for the automation.
2. Functions that are passed directly through lex to the output.
3. Actions from the input program, which appear as fragments of code. which are invoked by the automation simulator when needed.

To construct the automation, we begin by taking each regular expression pattern in the LEX program.

Steps to construct Automations

① we need a single automation that will recognize lexemes matching any of the patterns in the program,

② combine all the NFA's into one by introducing a new start state with ε-transitions to each of the start states of NFA's $N(P_i)$ for pattern P_i

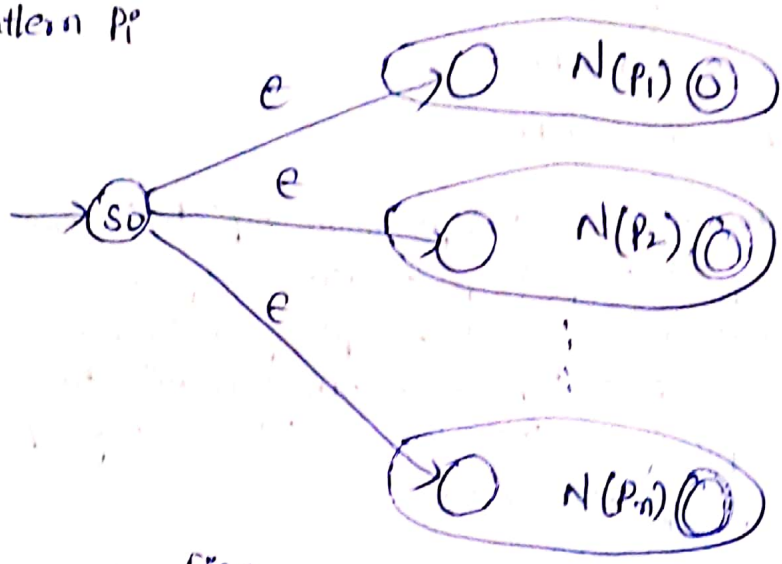


Fig: construction of NFA from LEX program.

eg: a {action A1 for pattern P_1}
abb {action A2 for pattern P_2}
a^*b^+ {action A3 for pattern P_3}

for string abb, pattern P_2 and pattern P_3 matches. But the pattern P_2 will be taken into account as it was listed first in LEX program.

for string aabbb ... matches pattern P_3 as it has many prefixes

The above figure shows NFAs for recognizing the above mentioned three patterns.

The combined NFA for all three given patterns is.

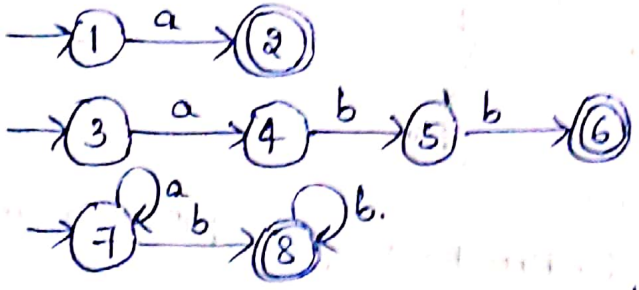


fig: NFA's for a, abb, a^*b^+

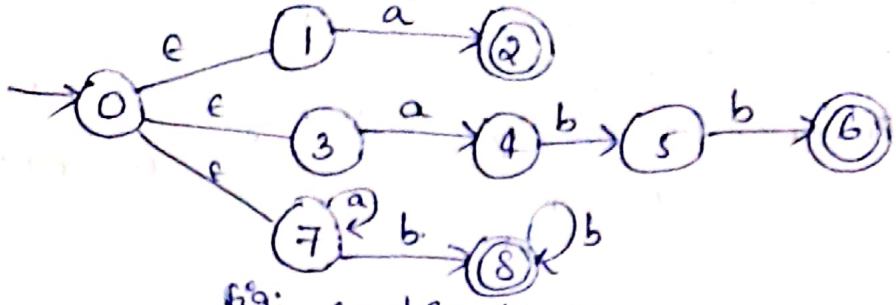


fig: combined NFA

② pattern matching Based on NFAs :-

Lexical Analyzer reads input from input buffer from the beginning of lexeme pointed by the pointer lexeme begin. forward pointer is used to move ahead of input symbols, calculates the set of states it is in at each point. If NFA simulation has no next state for some input symbol, then there will be no longer prefix which reaches the accepting state exists.

This process is repeated until one or more accepting states are reached. If there are several accepting states, then the pattern Pp which appears earliest in the list of LEX program is choosen.

③ DFA's for lexical analyzer :-

Another architecture, resembling the output of LEX, is to convert the NFA for all patterns into the equivalent DFA, using the subset construction.

Within each DFA state, if there are one or more accepting NFA states, determine the first pattern whose accepting state is represented & mark that pattern the output of the DFA state.

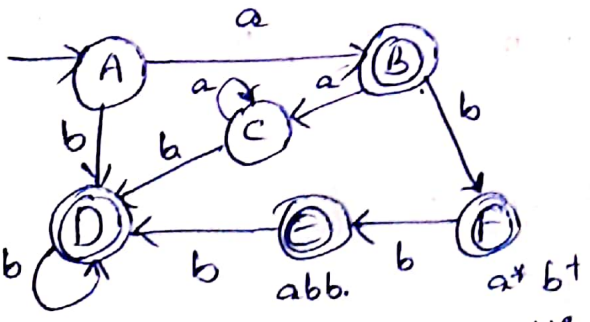


Fig: Transition graph for DFA handling the pattern a, abb & a*b+

=

④ Implementing the lookahead operator:-

The LEX lookahead operator "/" in a lex pattern r_1/r_2 is sometimes necessary, because the pattern "r_1" for a particular token may need to describe some trailing context r_2 in order to correctly identify the actual lexeme. "/" is treated as ϵ.

* The end occurs when the NFA enters a state such that

1. "s" has an ϵ-transition
2. There is a path from the start state of the NFA to state "s" that spell out x.
3. There is a path from state "s" to the accepting state that spells out y.
4. x is as long as possible for any xy satisfying conditions 1,3

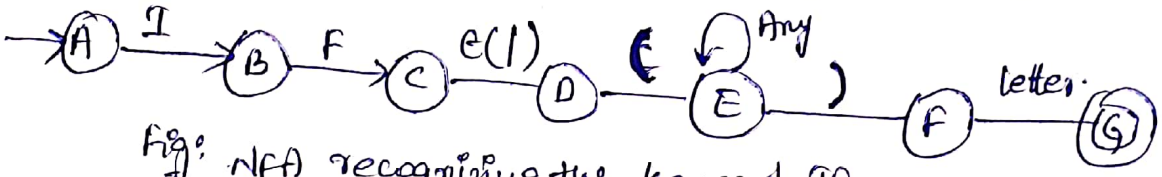


Fig: NFA recognizing the keyword IF.

=