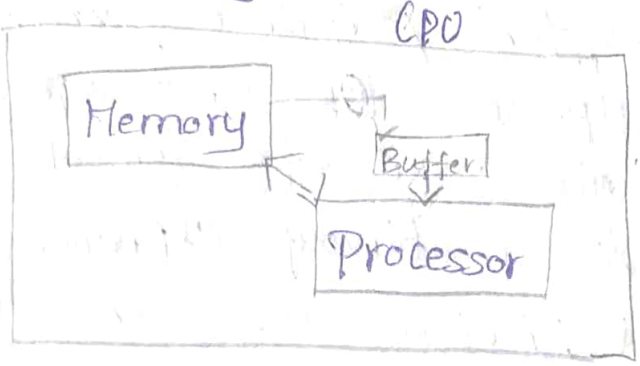
Input Buffering :



The Lexical Analyzer Scans the input String from left to right One Character at a time.

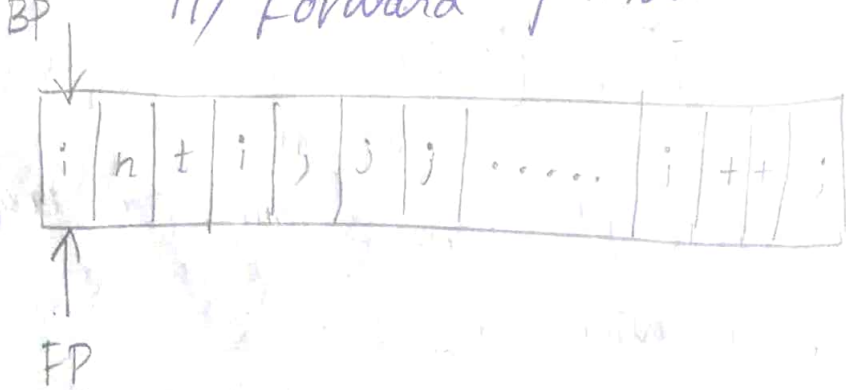
The Lexical Analyzer to access the Source Code directly from memory will take a Considerable amou of time.

Hence the Concept of Buffering is included so that a part of the Program can be brought into the buffer each time the lexier has to scan the Source Code.

In Input buffering we are using two Pointers

i) Begin Pointers.

ii) Forward Pointers.



Begin Pointer:

It is also known as lexeme begin. It always Point to the first Character of the lexeme.

Forward Pointer:

Initialy it points to the first Character of lexeme and then it moves forward until it reaches the delimeter Point.

All the Characters present between the begin pointer and forward pointer is Consider for Producing a token.

(X) Types of Input Buffering Scheme:

i) Single Buffer Scheme
ii) Buffer Pair Scheme
iii) Buffer Pair Scheme with Sentinels Characters.

i) Single Buffer Scheme:

↳ It is also known as one Buffer Scheme.

↳ In this Scheme it takes a single buffer of size 'N'. Where N holds 1024 bits to 4096 bits.

Drawback:

↳ If the length of the lexeme cross the boundary of buffer then the remaining part is to be loaded into the same buffer which will lead to over writing of the previous part of the lexeme.

ii) Buffer Pair Scheme:

↳ It is also known as two buffer scheme. In this scheme we are using two buffer b_1 and b_2 of same size 'N'. When we reach end of the buffer b_1 we can load the remaining part of the lexeme into the buffer b_2. Similarly if we reach the end of the buffer b_2 then the remaining part of the lexeme can be loaded into buffer b_1.

Algorithm:

Step 1 : FP := FP + 1

Step 2 : If FP = End B_1 then

Step 2.1 : Load buffer B_2.

Step 2.2 : FP := FP + 1;

Step3 : Else if FP = End of B_2 then

Step 3.1 : Load buffer B_1

Step 3.2 : FP := FP + 1

Step 4 : Else if FP = End of the file ($)

Step 4.1 : Stop lexical analysis

[End of if)

Buffer pair Scheme with Sentinals

Character:

In Case of buffer pair scheme each time FP moves on we need to test minimum two conditions to see whether we have reach end of buffer B_1 or end of the buffer B_2. In order to determine where to load the remaining part of the Lexeme.

The number of test Condition Can be reduced to one by inserting the Special Character Call Sentinal Character at each end of buffer that denotes EOf.

Algorithm:

Step 1: Fp := Fp+1

Step 2: If Fp = EOF, then

Step 2.1: If Fp = End of B_1, then

Step 2.1.1: load buffer B_2

Step 2.1.2: Fp = Fp+1

Step 2.2: Else if Fp = End of B_2, t

Step 2.2.1: load Buffer B,

Step 2.2.2: FP = Fp+1

[End of 2.1]

[End of 2]

Step 3: Else sto lexical analyz

[End of If)

Regular language:

A regular language over an alphabet is the one that can be obtained from the basic languages using the operations Union, Concatenation, Kl

A language is said to be regular language if there exists a deterministic finite automata (DFA) for that language.

The language accepted by DFA is a regular language.

A regular language can be converted into a regular expressions by leaving out {} or replacing {} with () and by replacing ∪ by +.

Regular expression:

Regular expression is a formula that describes a possible set of strings.

Properties of Regular expression:

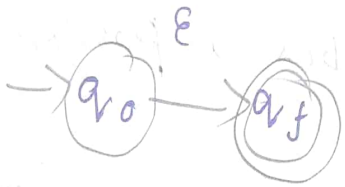
	Axiom	Description
1.	$r \mid s = s \mid r$	$\mid$ is Commutative
2.	$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is Associative
3.	$(rs)t = r(st)$	Concatenation is associative.
4.	$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$	Concatenation distributive over $\mid$

5)	$r\varepsilon = r$ $\varepsilon r = r$	ε is the identity for Concatenation
6)	$(r\|\varepsilon)^* = r^*$	-
7)	$r^{**} = r^*$	* is idempotent

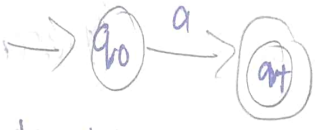
Regular expression to finite automata

To Construct NFA from regular expression the thomson Construction method is used

1) When $r = \varepsilon$

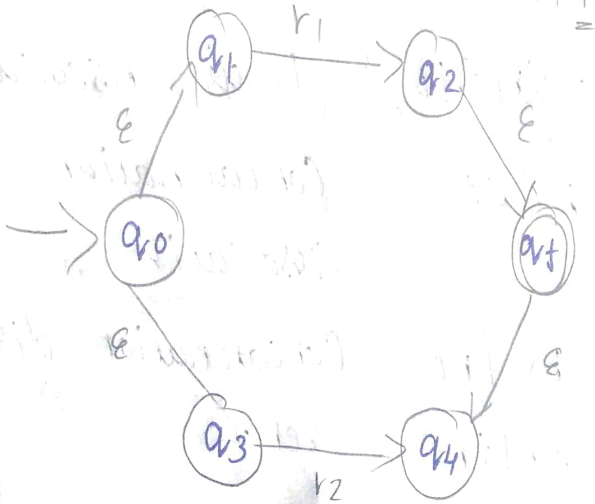


2) When $r = a$



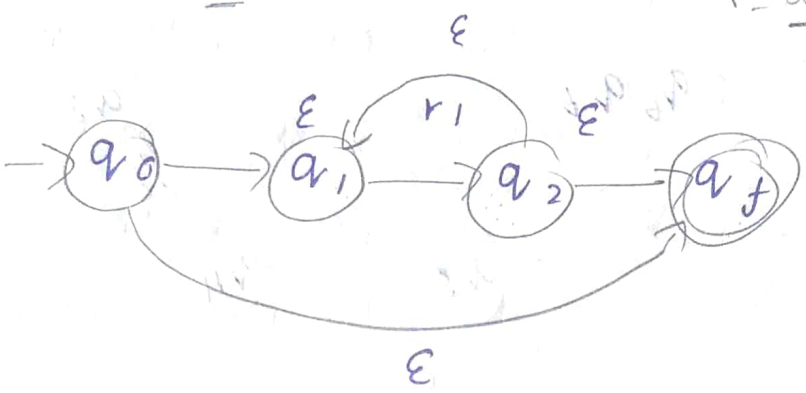
3) When $r = r_1 + r_2$

$r_1 | r_2$

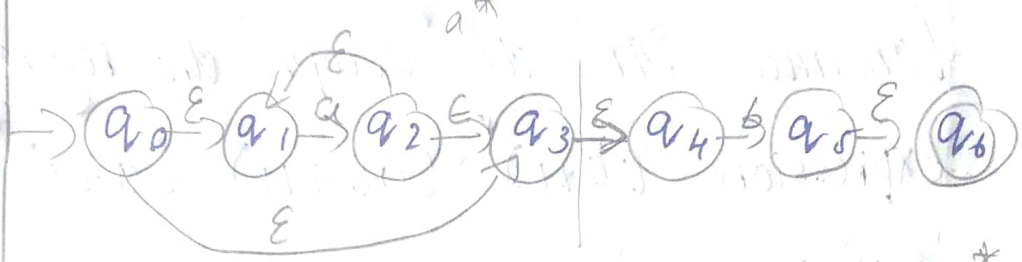


4) When $r = r_1 \cdot r_2$

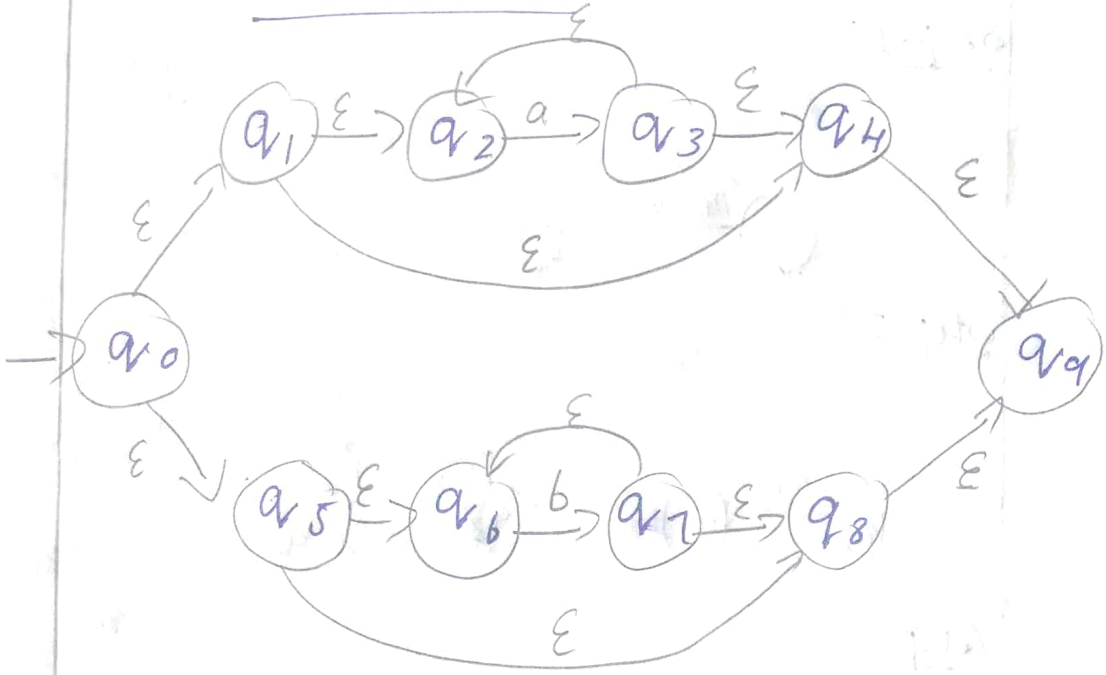
5) When $r = r_1^*$



Construct NFA for $r = a^* b$

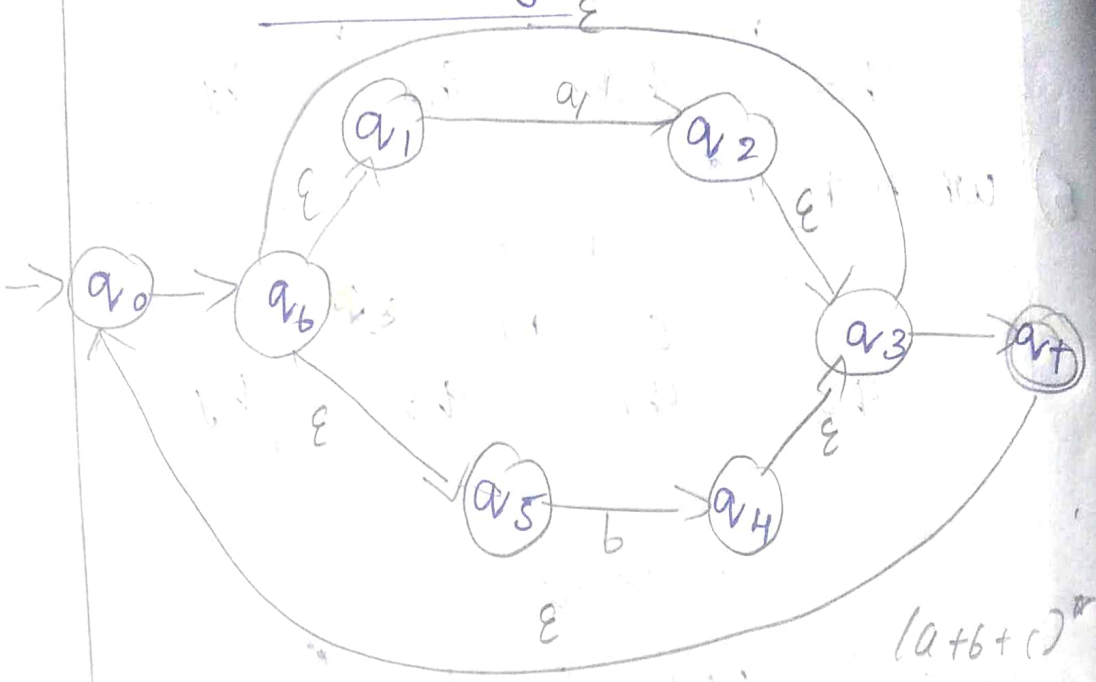


Construct NFA for $r = a^* / b^*$ $a^* + b^*$



aa+bb

Construct NFA for $r = (a+b)^*$



$(a+b+\epsilon)^*$

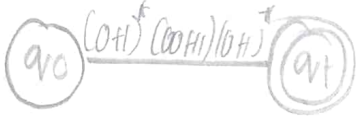
(Discreate Finite Automata)

Construct DFA to accept the regular expression $(0+1)^*(00+11)(0+1)^*$

Solution:

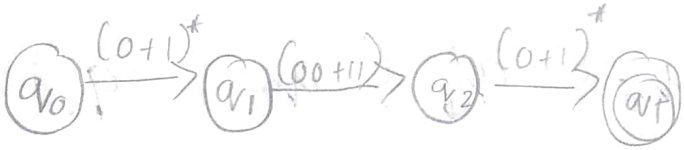
Step-1:

Starting state and ending state



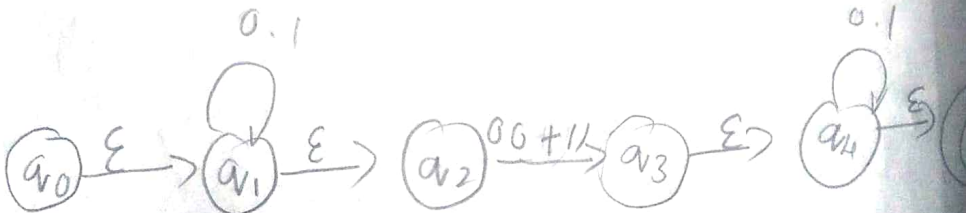
Step-2:

Split the term

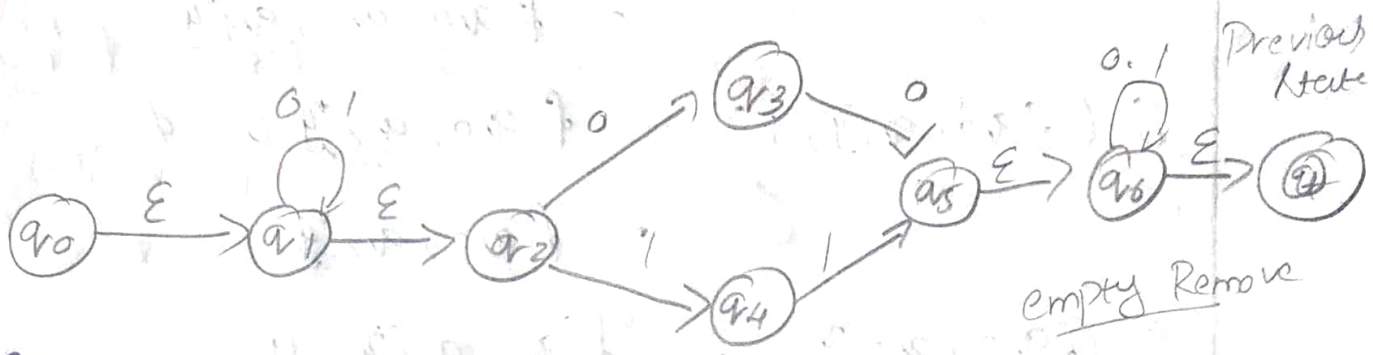


More than one

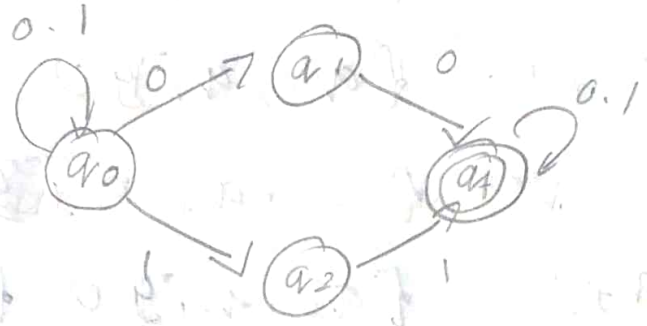
Step-3:



Step-4:



Step-5:



Step-6:

Draw a transion table

	0	1
q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$
q_1	$\{q_f\}$	-
q_2	-	$\{q_f\}$
q_f	$\{q_f\}$	$\{q_f\}$

Step-7:

$\delta[q_0, 0) = [q_0, q_1]$ → new

$\delta(q_0, 1) = [q_0, q_2]$ → new

$\delta(q_1, 0) = [q_f]$

$\delta(q_1, 1) = \phi$

$\delta(q_2, 0) = \phi$

$\delta(q_2, 1) = [q_f]$

$\delta(\{q_0, q_1\}, 0) = \delta(q_0, 0) \cup \delta(q_1, 0)$

$= [q_0, q_1] \cup [q_f]$

$= [q_0, q_1, q_f]$ → new

$\delta(\{q_0, q_1\}, 1) = [q_0, q_2] \cup \phi$

$= [q_0, q_2]$

$\delta(\{q_0, q_2\}, 0) = [q_0, q_1] \cup \phi$

$= [q_0, q_1]$

$\delta(\{q_0, q_2\}, 1) = [q_0, q_2] \cup [q_f]$

$= [q_0, q_2, q_f]$ → new

$\delta(\{q_0, q_1, q_f\}, 0) = [q_0, q_1] \cup [q_f] \cup [q_1]$

$= [q_0, q_1, q_f]$

$\delta(\{q_0, q_1, q_f\}, 1) = [q_0, q_2] \cup \phi \cup [q_f]$

$= [q_0, q_2, q_f]$

$\delta(\{q_0, q_2, q_f\}, 0) = [q_0, q_1] \cup \phi \cup [q_f]$

$= [q_0, q_1, q_f]$

$\delta(\{q_0, q_2, q_f\}, 0) = [q_0, q_2] \cup [q_f] \cup [q_1]$

$= [q_0, q_2, q_f]$

There is no new state.

Step-8:

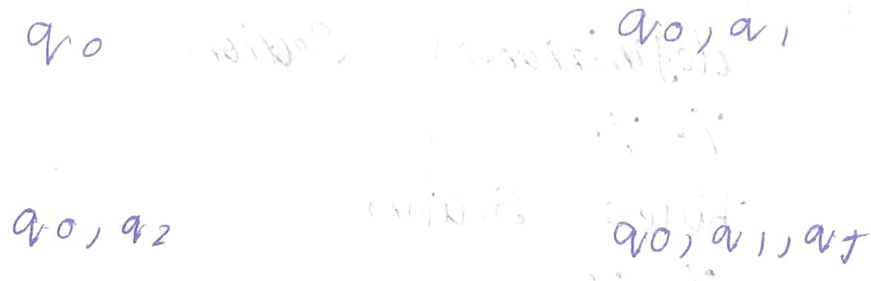
	0	1
→ $[q_0]$	$[q_0, q_1]$	$[q_0, q_2]$
$[q_1]$	$[q_f]$	q
$[q_2]$	q	$[q_f]$
$[q_f]$	$[q_f]$	$[q_f]$
$[q_0, q_1]$	$[q_0, q_1, q_f]$	$[q_0, q_2]$
$[q_0, q_2]$	$[q_0, q_1]$	$[q_0, q_2, q_f]$
$[q_0, q_1, q_f]$	$[q_0, q_1, q_f]$	$[q_0, q_2, q_f]$ } same
$[q_0, q_2, q_f]$	$[q_0, q_1, q_f]$	$[q_0, q_2, q_f]$

Step-9:

The minimize of DFA will be,

	0	1
q_0	$[q_0, q_1]$	$[q_0, q_2]$
$[q_0, q_1]$	$[q_0, q_1, q_f]$	$[q_0, q_2]$
$[q_0, q_2]$	$[q_0, q_1]$	$[q_0, q_2, q_f]$
$[q_0, q_1, q_f]$	$[q_0, q_1, q_f]$	$[q_0, q_2, q_f]$

Step-10: Transaution diagram :



7.

(6 marks)

Lex tool

Yacc tool

Lex tool: [Lexical Analyzer Generator]

↳ Lex is a lexical Analyzer tool mostly used with Yacc.

↳ It is a tool for regonizing tokens in a Program.

↳ Tokens are the terminals of a language. In a Programming language

Programing Language: Identifiers, Operators, Keywords are the tokens. The regular expressions defined the tokens. It lexically analysis (matches) the Patterns (regular expression) given as a input String or as a file.

Structure of lex Program:

↳ The lex Program consists of three Sections Seperated by a line with just %.%.

Format:

definitions Section
%.%.
Rules Section
%.%.

Auxillary function /

User Code Section

Definition Section:

↳ Generally used to declare functions, include header files or defined a global variables and constants.

↳ The text is enclosed in %{ %} curly brackets.

→ Anything returns in this bracket is copied directly to the file lex.yy.c.

Example:

```
%{
    # include < stdio.h>
    int global - variable;
    %}
```

Rules Section:

↳ Each rule has the form, Pattern action.

i) Pattern: Where pattern describes a pattern to be matched on the input, action describes what action to be performed.

ii) Action:

If the action is empty the match token is discarded.

Example:

```
%. %.
{digit}+ {Printf ("number"); }
{letter}* {Printf ("name"); }
%. %.
```

Here,

The digit denotes the value [0-9]

The letter denotes the value [a-z, A-Z]

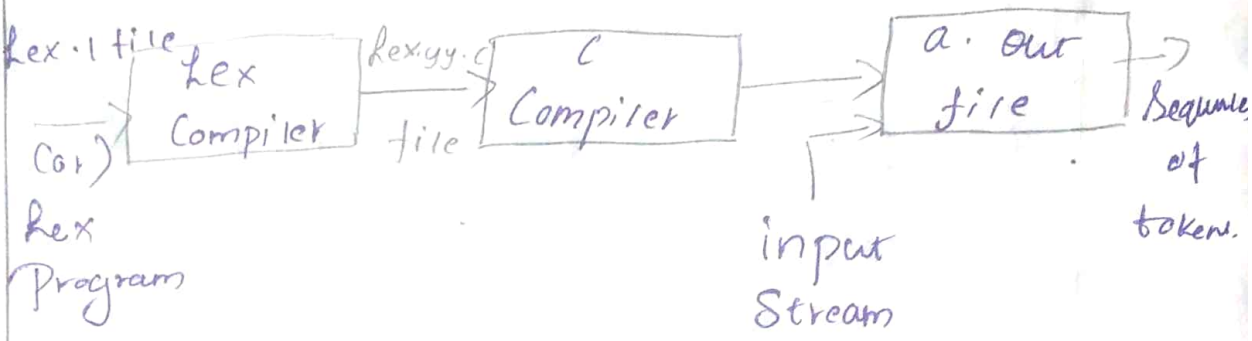
Auxillary Function:

↳ The lex generates C code for the rules specified in the rule section and places the code into a single function called yylex().

Example:

```
/* Declarations */
%%
/* Rules */
%%
int main()
{
    yylex();
    return 1;
}
```

Processing a lex Program:



Program to Check whether a given number is even or odd:

```
%option noyywrap
%{
#include <stdio.h>
int i;
%}
%%
[0-9]+      {i=atoi(yytext);
            if(i%2==0)
            Printf("even no");
            else
```

```
Printf ("odd no");
};
*/ */
int main ()
{
    int x ();
    return 0;
}
```