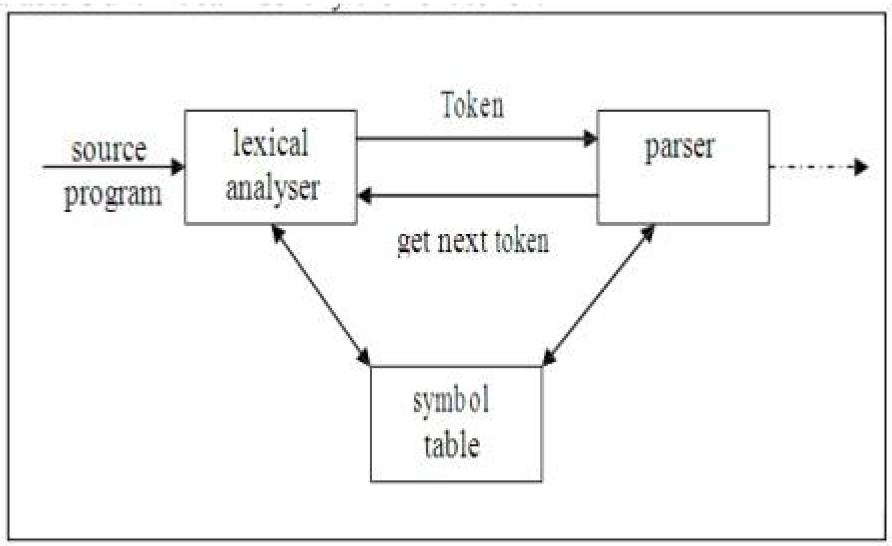
LEXICAL ANALYSIS

- Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens.
- A program which performs lexical analysis is termed as a lexical analyzer (lexer), tokenizer or scanner.
- The sequence of tokens produced by lexical analyzer helps the parser in analyzing the syntax of programming languages.
- The input to a lexical analyzer is the pure high-level code from the preprocessor. It identifies valid lexemes from the program and returns tokens to the syntax analyzer, one after the other, corresponding to the ***getNextToken*** command from the syntax analyzer.



Main of task of lexical analysis

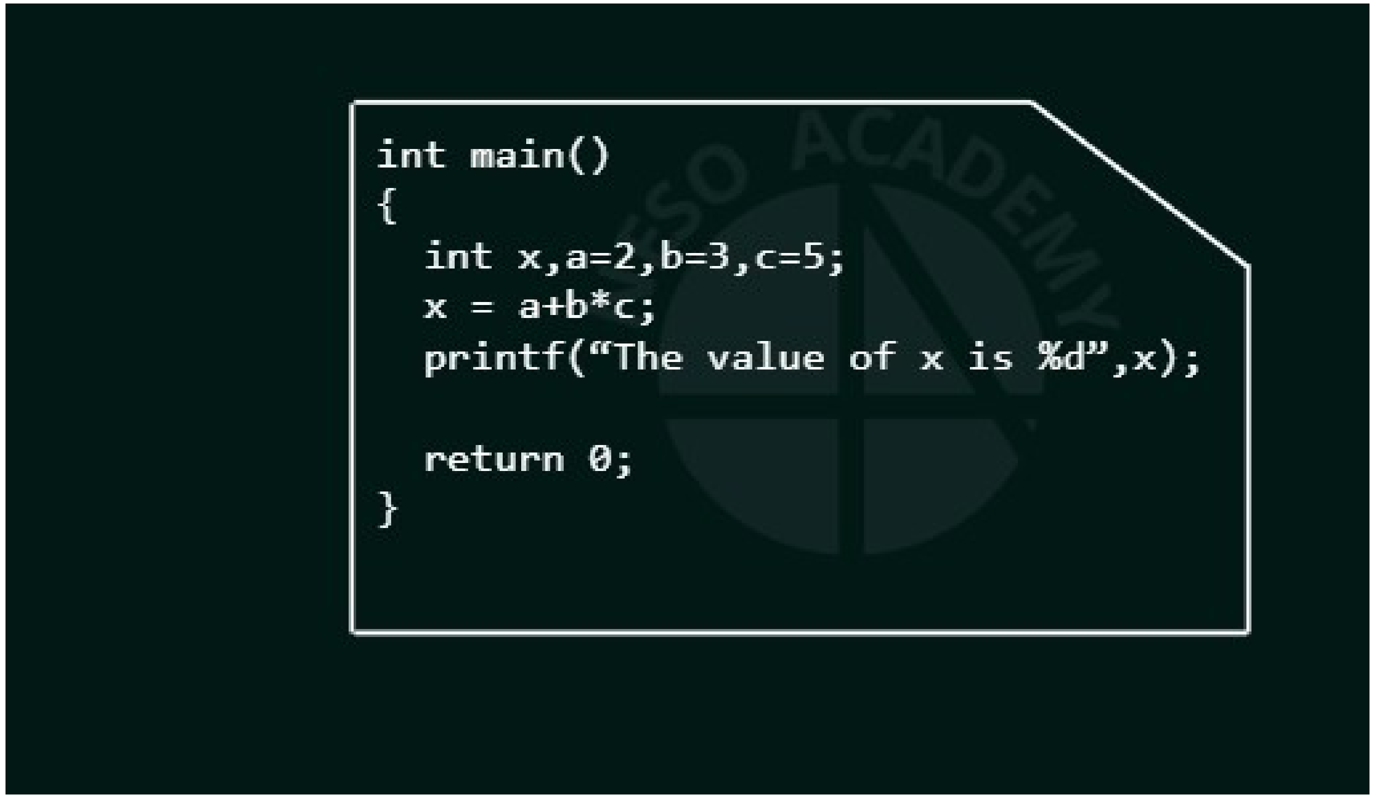
- Scans the pure HLL code line by line
- Takes lexemes as i/p and produces Tokens
- Removes comments and whitespaces(blank space (' '), newline ('\n'), horizontal tab ('\t'), carriage return ('\r'), form feed ('\f') or vertical tab ('\v’))from the pure HLL code.
- Find lexical errors
- Return the Sequence of valid tokens to the syntax analyzer
- When it finds an identifier, it has to make an entry into the symbol table.

Contd…

- **There are three important terms :**

1. **Tokens**: A Token is a pre-defined sequence of characters that cannot be broken down further. It is like an abstract symbol that represents a unit. A token can have an optional attribute value. There are different types of tokens:
    1. Identifiers (user-defined)
    2. Punctuations (;, ,, {}, etc.)
    3. Operators (+, -, *, /, etc.)
    4. Special symbols
    5. Keywords
    6. Constant
    7. Literals
2. **Lexemes**: A lexeme is a sequence of characters matched in the source program that matches the pattern of a token. **For example**: (,) are lexemes of type punctuation where punctuation is the token.
3. **Patterns:** A pattern is a set of rules a scanner follows to match a lexeme in the input program to identify a valid token. It is like the lexical analyzer's description of a token to validate a lexeme. **For example**, the characters in the keyword are the pattern to identify a keyword. To identify an identifier the pre-defined set of rules to create an identifier. i.e., pattern of an identifier is letter followed by letter or digits.

Contd…..



```
int main()
{
  int x,a=2,b=3,c=5;
  x = a+b*c;
  printf("The value of x is %d",x);

  return 0;
}
```

Contd….

1. The number of tokens in the following C statement is

```
printf("HELLO WORLD"); -> 5
```

2. (GATE2000). The number of tokens in the following C statement is

```
printf("i=%d, &i=%x", i, &i);
```

3.

Specification of a token:

- A token is the smallest individual element of a program that is meaningful to the compiler. It cannot be further broken down.
- Regular expression are used to specify tokens.
- Regular expression generates regular language. Language means a set of strings. Strings means a set of alphabet.
- So in this concept , we will discuss about

1. Alphabet
2. String
3. Language
4. Operation on language
5. Regular expression
6. Regular Definition

1. Alphabet

- It is a set symbol denoted by ∑.

Example:

$\sum = \{0, 1\}$ It is a binary set.

$\sum = \{a, b, \ldots\ldots, z\}$ It is a set of lower case letter.

2. String

- It is finite set of symbols generated from ∑ (alphabet)
- Example:

  ∑={a,b}

  From this we can generate n no. of strings like a, ab,aa,ba………..

- **Length of String**
- The length of the string can be determined by the number of alphabets in the string. The string is represented by the letter '*s*' and |*s*| represents the length of the string. Let's consider the string:
- *s* = banana

|*s*| = 6

- **Note**: The empty string or the string with length 0 is represented by ' ∈ '.

Term related to string

1. Prefix of String

- The prefix of the string is the preceding symbols present in the string and the string *s* itself.
- *For example:*
- *s* = abcd
- The prefix of the string abcd: $\in$, a, ab, abc, abcd

2. **Suffix of String**

- Suffix of the string is the ending symbols of the string and the string *s* itself.
- *For example:*
- *s* = abcd
- Suffix of the string abcd: $\in$, d, cd, bcd, abcd

Contd….

3. Proper Prefix of String

- The proper prefix of the string includes all the prefixes of the string excluding $\in$ and the string *s* itself.
- Proper Prefix of the string abcd: a, ab, abc

4. Proper Suffix of String

- The proper suffix of the string includes all the suffixes excluding $\in$ and the string *s* itself.
- Proper Suffix of the string abcd: d, cd, bcd

5. Substring of String

- The substring of a string *s* is obtained by deleting any prefix or suffix from the string.
- Substring of the string abcd: $\in$, abcd, bcd, abc, …

Contd…

6. Proper Substring of String

- The proper substring of a string *s* includes all the substrings of *s* excluding $\in$ and the string *s* itself.
- Proper Substring of the string abcd: bcd, abc, cd, ab…

7. Subsequence of String

- The subsequence of the string is obtained by eliminating zero or more (not necessarily consecutive) symbols from the string.
- A subsequence of the string abcd: abd, bcd, bd, …

8. Concatenation of String

- If *s* and *t* are two strings, then *st* denotes concatenation.
- *s* = abc *t* = def
- Concatenation of string *s* and *t* i.e. *st* = abcdef

3. Language

- Language is a set of strings over some fixed alphabets. Like the English language is a set of strings over the fixed alphabets ‘a to z’.

Operation on language

1. Union

- Union is the most common set operation. Consider the two languages L and M. Then the union of these two languages is denoted by:
- L ∪ M = { *s* | *s* is in L or *s* is in M}
- That means the string *s* from the union of two languages can either be from language L or from language M.
- If L = {a, b} and M = {c, d}Then L ∪ M = {a, b, c, d}

2. Concatenation

- Concatenation links the string from one language to the string of another language in a series in all possible ways. The concatenation of two different languages is denoted by:
- L・M = {*st* | *s* is in L and *t* is in M}If L = {a, b} and M = {c, d}
- Then L・M = {ac, ad, bc, bd}

Contd….

3. Kleene Closure

- Kleene closure of a language L provides you with a set of strings. This set of strings is obtained by concatenating L zero or more time. The Kleene closure of the language L is denoted by:
- If L = {a, b}L^* = { $\in$, a, b, aa, bb, aaa, bbb, …}

4. Positive Closure

- The positive closure on a language L provides a set of strings. This set of strings is obtained by concatenating 'L' one or more times. It is denoted by:
- It is similar to the Kleene closure. Except for the term L^0, i.e. L^+ excludes $\in$ until it is in L itself.
- If L = {a, b}L^+ = {a, b, aa, bb, aaa, bbb, …}
- So, these are the four operations that can be performed on the languages in the lexical analysis phase.

5. Regular Expression

1. If regular expression (R) is equal to Epsilon (ε) then language of Regular expression (R) will represent the epsilon set i.e. { ε}. Mathematical equation is given below,

$$\text{If } R = \varepsilon \text{ Then } L(R) = \{\varepsilon\}$$

regular expression (R) is equal to Epsilon (ε)

2. If regular expression (R) is equal to Φ then language of Regular expression (R) will represent the empty set i.e. { }. Mathematical equation is given below,

$$\text{If } R = \Phi \text{ Then } L(R) = \{\ \}$$

empty regular expression

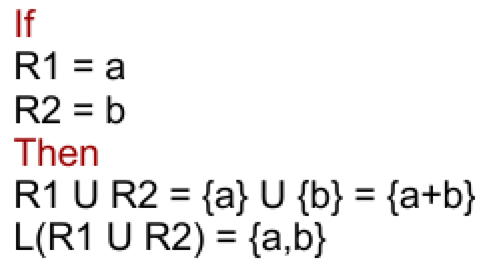
3. If regular expression (R) is equal to a input symbol “a” which belongs to sigma, then language of Regular expression (R) will represent the set which having “a” alphabet i.e. {a}. Mathematical equation is given below,

$$\text{If } R = a \text{ Then } L(R) = \{a\}$$

regular expression (R) is equal to one input

Contd….

4. Union of Two Regular Expressions will always produce a regular language. Suppose R1 and R2 are two regular expressions. IF R1= a, R2=b then R1 U R2 =a+b So L(R1 U R2) = {a,b}, still string “a,b” is a regular language.

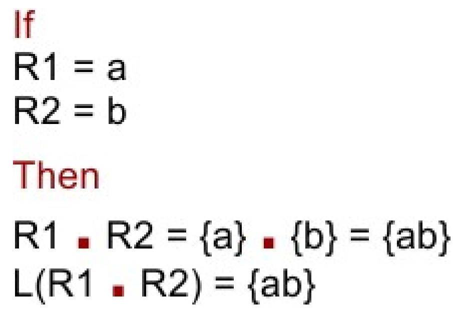


If
R1 = a
R2 = b
Then
R1 U R2 = {a} U {b} = {a+b}
L(R1 U R2) = {a,b}

Hence, above equation shows that {a,b} is also a regular language.

Contd….

5. Concatenation of two Two Regular Expressions will always produce a regular language. IF R1= a, R2=b then R1.R2 =a.b So L(R1.R2) = {ab}, still string “ab” is a regular language.



If
R1 = a
R2 = b

Then

R1 . R2 = {a} . {b} = {ab}
L(R1 . R2) = {ab}

Hence, above equation shows that {ab} is also a regular language.

Contd….

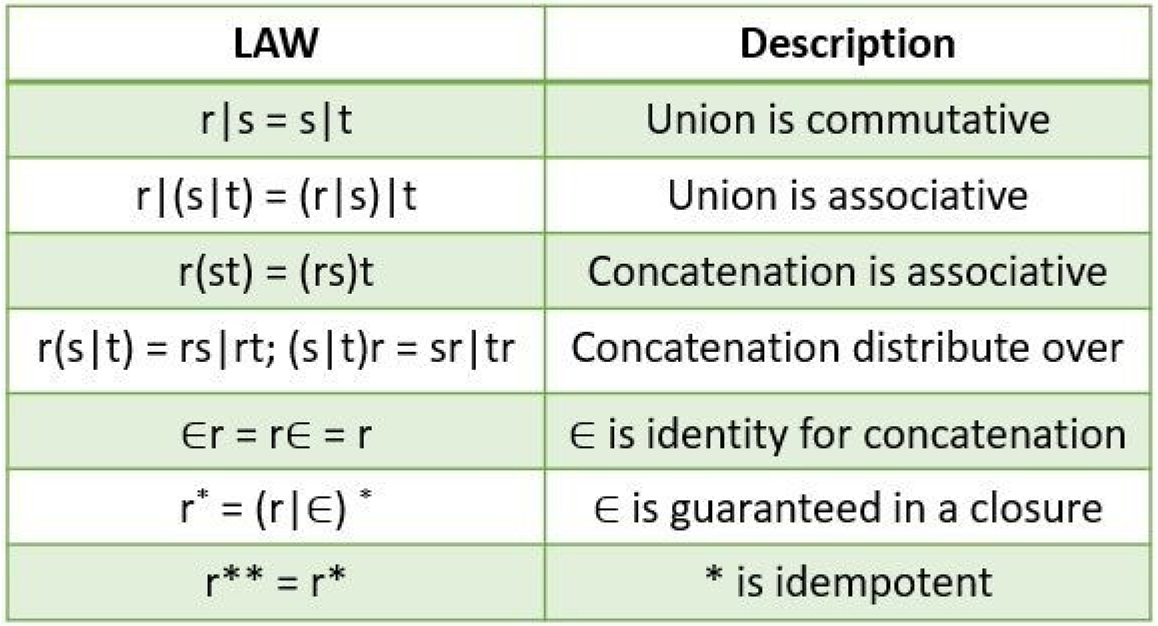
6. Kleene closure of Regular Expression (RE) is also a regular language

If R1 = x and (R1)* is still a regular language

In a regular expression, x* means zero or more occurrence of x. It can generate { **ε**, x, xx, xxx, xxxx, …..}

In a regular expression, x^+means one or more occurrence of x. It can generate {x, xx, xxx, xxxx, …..}

Consider the regular expressions *r*, *s*, *t*. Algebraic laws for these regular expressions.



LAW	Description
$r \mid s = s \mid t$	Union is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	Union is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s \mid t) = rs \mid rt; (s \mid t)r = sr \mid tr$	Concatenation distribute over
$\in r = r\in = r$	$\in$ is identity for concatenation
$r^* = (r \mid \in)^*$	$\in$ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Algebraic Law for Regular Expression

Notations

1. The ‘+’ symbol denotes one or more occurrence
2. The ‘*’ symbol denotes zero or more occurrence
3. 2. The `?` symbol denotes zero or one occurrence
4. 3. The `[ ]` symbol denotes character classes

Regular Definition

Giving names to regular expressions is referred to as a Regular definition.

If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

dl → r 1

d2 → r2

………

dn → rn

1.Each di is a distinct name.

2.Each ri is a regular expression over the alphabet Σ U {dl, d2,. . . , di-l}.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:

letter → A | B | …. | Z | a | b | …. | z |

digit → 0 | 1 | …. | 9

id → letter (letter | digit) *

Or

id → [a-zA-Z_][a-z A-Z 0-9]*

Contd….

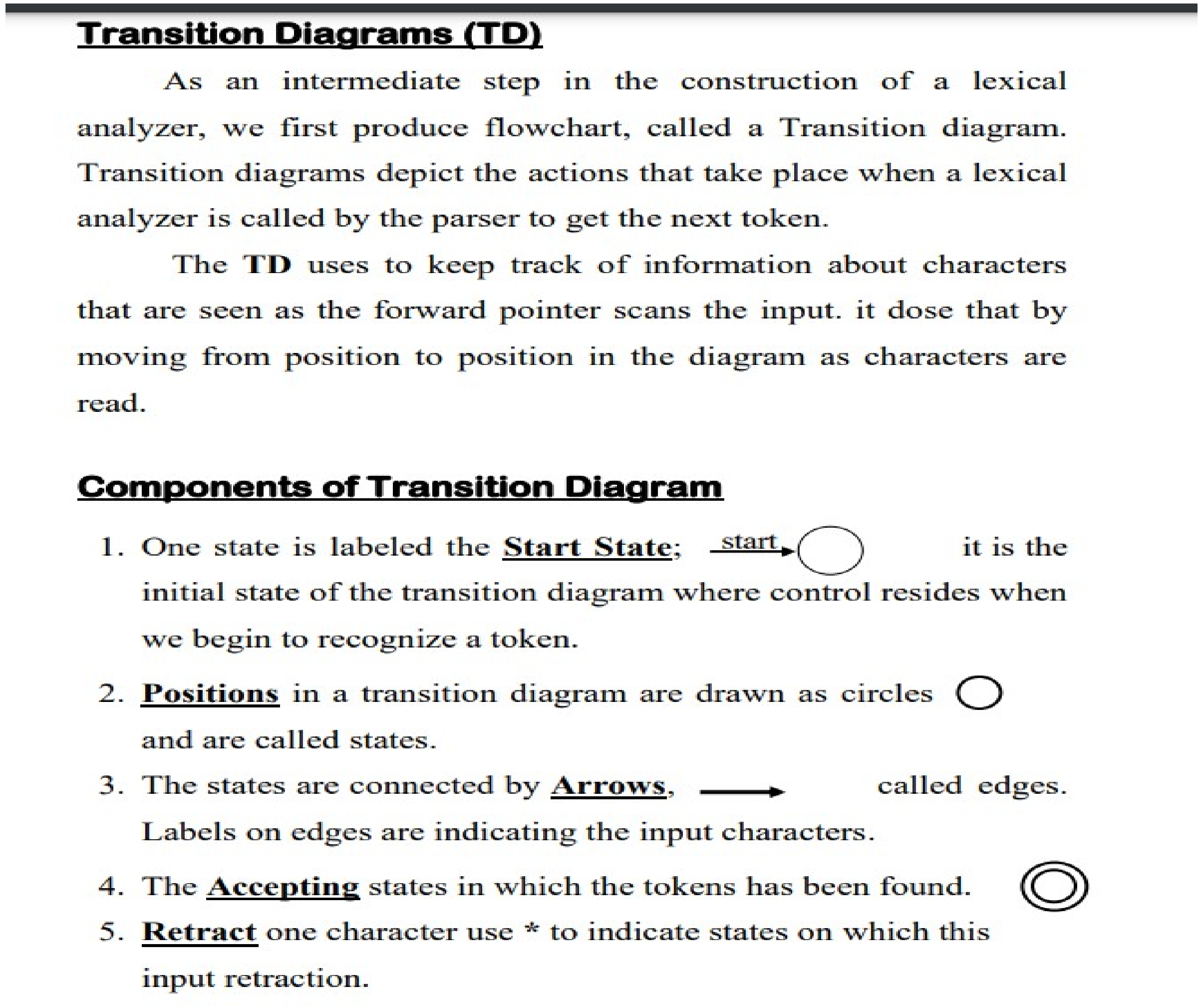
The regular expression number can be written using the regular definition as;

123 , 456
123.45
123.45E23
123.45E-23

digit $\rightarrow$ 0 | 1 | 2 | … | 9
digits $\rightarrow$ digit+
optional_fraction $\rightarrow$ (. digits) ?
optional_exponent $\rightarrow$ (E (+ | -) ? digits) ?
num $\rightarrow$ digits optional_fraction optional_exponent

Recognition of tokens

- Tokens are recognized using transition diagram



Transition Diagrams (TD)

As an intermediate step in the construction of a lexical analyzer, we first produce flowchart, called a Transition diagram. Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token.

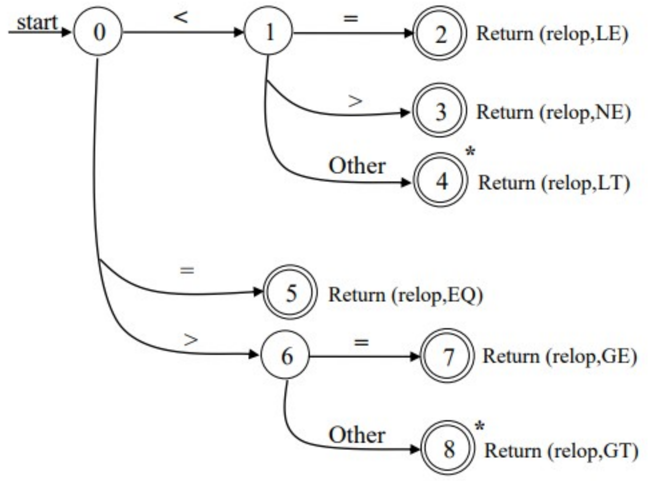
The **TD** uses to keep track of information about characters that are seen as the forward pointer scans the input. it dose that by moving from position to position in the diagram as characters are read.

Components of Transition Diagram

1. One state is labeled the **Start State**; $\xrightarrow{\text{start}}\bigcirc$ it is the initial state of the transition diagram where control resides when we begin to recognize a token.
2. **Positions** in a transition diagram are drawn as circles $\bigcirc$ and are called states.
3. The states are connected by **Arrows**, $\longrightarrow$ called edges. Labels on edges are indicating the input characters.
4. The **Accepting** states in which the tokens has been found. $\circledcirc$
5. **Retract** one character use * to indicate states on which this input retraction.

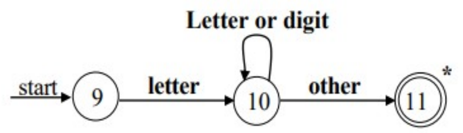
Examples:

1. Recognition of relational operator



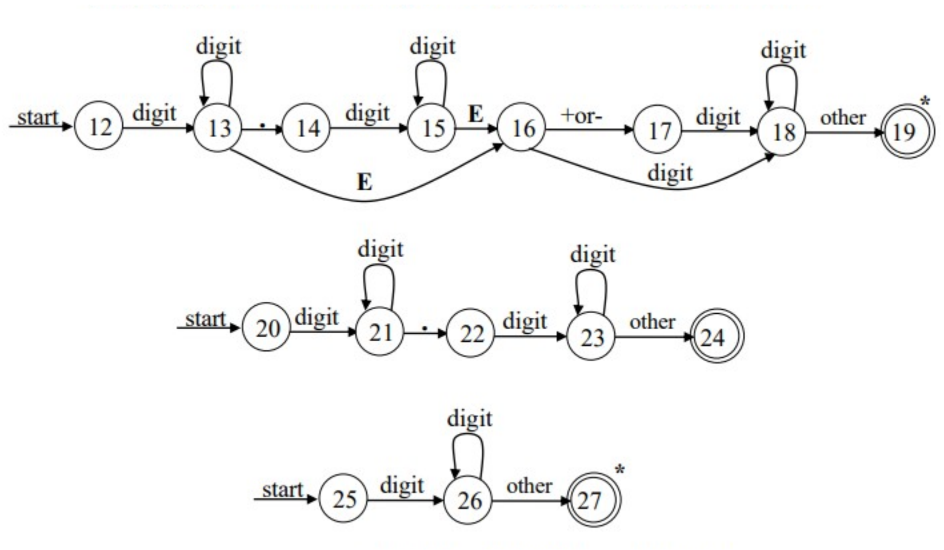
Transition Diagram for relation operators

2. Recognition of an identifier or keyword



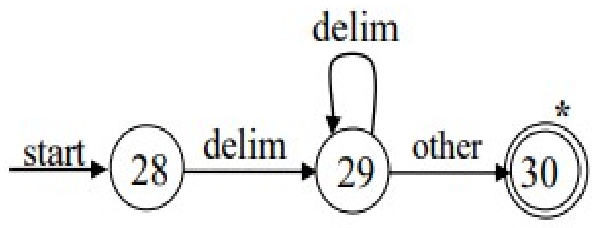
Transition Diagram for identifiers and keywords

3. Recognition of a number



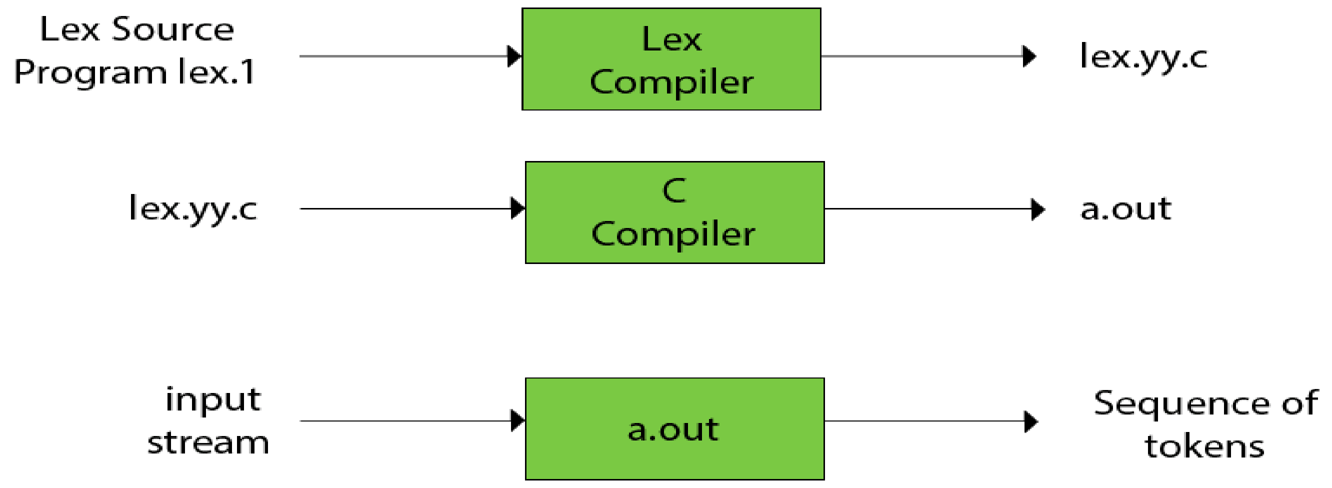
4. Recognition of delimiter

ws → delimiter(delimeter) *


$$\begin{tikzcd}
\arrow[r, "start"] & 28 \arrow[r, "delim"] & 29 \arrow[loop above, "delim"] \arrow[r, "other"] & 30^{*}
\end{tikzcd}$$

Lex and flex Tool

- It is language or tool for specifying lexical analyzer.
- Basically, Lex and Flex are lexical analyzer generators
- Lex and Flex are good at matching patterns
- Lex was originally written by Mike Lesk and Eric Schmidt in 1975
- Flex is an open-source alternative to Lex



Working/Function of Lex tool

- Source code is in lex language with filename.l extension. It is given to lex compiler which is the lex tool and produces lex.yy.c , which is a c program.
- C compiler then run this c code i.e., lex.yy.c program and produces an output a.out , which is a lexical analyzer.
- a.out transforms an input stream into a sequence of token.

lex.yy.c : C program/ c language file
File.l : Lex source program file will always have .l extension
a.out : Lexical Analyzer

Note: Lex language is used to program in the lex tool.

Structures of a lex program

- A LEX program consists of three sections : Declarations, Rules and Auxiliary functions.

```
DECLARATIONS
%%
TRANSLATION RULES
%%
AUXILIARY FUNCTIONS
```

1. Declaration

This section includes declaration of variables, constants and regular definitions. The declarations section can be empty.

```
%{
Declaration
%}
```

Example:

```
%{
 int a , b;
float count=0;
%}
digit [0-9]
Letter [a-z A-Z]
```

2. Translation Rules:

- Starts with %% and end with %% .
- Every Rule is specify in the form of pattern followed by action

  pattern {action}

- Each pattern is a regular expression which may use regular definitions defined in the declarations section.
- Each action is a fragment of C-code.
- Pattern and action are separated by as ingle line or multiples whitespaces such as blank space and tap space.

3. Auxiliary Function:

- All necessary function are defined here.

- Following are the lex predefined functions and variables :-
- **yyin** :- the input stream pointer (i.e it points to an input file which is to be scanned or tokenised), however the default input of default main() is stdin .
  yylex() :- implies the main entry point for lex, reads the input stream generates tokens, returns zero at the end of input stream . It is called to invoke the lexer (or scanner) and each time yylex() is called, the scanner continues processing the input from where it last left off.
  yytext :- a buffer that holds the input characters that actually match the pattern (i.e lexeme) or say a pointer to the matched string .
  yyleng :- the length of the lexeme .
  yylval :- contains the token value .
  yyval :- a local variable .*
  yyout :- the output stream pointer (i.e it points to a file where it has to keep the output), however the default output of default main() is stdout .
  yywrap() :- it is called by lex when input is exhausted (or at EOF). default yywrap always return 1.
  yymore() :- returns the next token .
  yyless(k) :- returns the first k characters in yytext .
  yyparse() :- it parses (i.e builds the parse tree) of lexeme .*

```
%{
#include <stdio.h>
%}

%%
"Hi"  {printf("Hello");}
 .* {printf ("Wrong");}
%%
  int main()
{
   printf(" Enter the input:\n");
    yylex();
}
 int yywrap(){ return 1; }
```

```
%{
#include<stdio.h>
int i;
%}
%%
[0-9]+     {i=atoi(yytext);
          if(i%2==0)
               printf("The given no. is even.\n");
          else
         printf("The given no. is odd.\n");}
%%
   int yywrap(){}
   int main()
{
   printf("Enter the number:\n");
    yylex();
    return 0;
}
```

Compiler construction toolkit

- Compiler Construction Tools are specialized tools that help in the implementation of various phases of a compiler. These tools help in the creation of an entire compiler or its parts.
- Some of the commonly used compiler constructions tools are:-
    1. Parser Generator
    2. Scanner Generator
    3. Syntax Directed Translation Engines
    4. Automatic Code Generators
    5. Data-Flow Analysis Engines

Contd…

1. Scanner Generators

- **Input:** Regular expression description of the tokens of a language
  Output: Lexical analyzers.
  Scanner generator generates lexical analyzers from a regular expression description of the tokens of a language.

2. Parser Generators

- **Input:** Grammatical description of a programming language
  Output: Syntax analyzers.
- Parser generator takes the grammatical description of a programming language and produces a syntax analyzer.

Contd….

3. Syntax-directed Translation Engines

- **Input:** Parse tree.
  Output: Intermediate code.
  Syntax-directed translation engines produce collections of routines that walk a parse tree and generates intermediate code.

4. Automatic Code Generators

- **Input:** Intermediate language.
  Output: Machine language.
  Code-generator takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for a target machine.

5. Data-flow Analysis Engines

- Data-flow analysis engine gathers the information, that is, the values transmitted from one part of a program to each of the other parts. Data-flow analysis is a key part of code optimization.

6. Compiler Construction Toolkits

- The toolkits provide integrated set of routines for various phases of compiler. Compiler construction toolkits provide an integrated set of routines for construction of phases of compiler.

Contd….

What is the purpose of compiler construction tools?

- The purpose of compiler construction tools is to help with the development of compilers and other language processing tools. These tools assist in different stages of the compiler development process, such as lexical analysis, parsing, optimization, etc.