1) Role of lexical analysis and its issues.

- ✓ The lexical analyzer is the first phase of compiler.
- ✓ Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
- ✓ It may also perform secondary task at user interface.
- ✓ One such task stripping out from the source program comments and white space in the form of blanks, tab, and newline characters.
- ✓ Some lexical analyzer are divided into cascade of two phases, the first called scanning and second is "lexical analysis".
- ✓ The scanner is responsible for doing simple task while lexical analysis does the more complex task.

Issues in Lexical Analysis:

There are several reason for separating the analysis phase of compiling into lexical analysis and parsing:

- ✓ Simpler design is perhaps the most important consideration. The separation of lexical analysis often allows us to simplify one or other of these phases.
- ✓ Compiler efficiency is improved.
- ✓ Compiler portability is enhanced.

2) Explain token, pattern and lexemes.

Token: Sequence of character having a collective meaning is known as *token*.

Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

Lexeme: The character sequence forming a token is called *lexeme* for token.

Pattern: The set of rules by which set of string associate with single token is known as *pattern*

Token	Lexeme	Pattern
id	x y n0	letter followed by letters and digits
number	3.14159, 0, 6.02e23	any numeric constant
If	If	if
relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
Literal	"abc xyz"	anything but ", surrounded by " 's

1. **if(x<=5) :**Token – if (keyword),
   X (id),
   <= (relation),
   5 (number)

Lexeme - if , x ,<=, 5

2. total = sum + 12.5

Token – total (id),
= (relation),
Sum (id),
+ (operator)
12.5 (num)
Lexeme – total, =, sum, +, 12.5

3) What is input buffering? Explain technique of buffer pair.

The speed of lexical analysis is a concern in compiler design.

1. Buffer pair:

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.

Two pointers to the input are maintained:

1. Pointer *Lexeme Begin*, marks the beginning of the current lexeme, whose extent we are attempting to determine
2. Pointer *Forward,* scans ahead until a pattern match is found.

code to advance forward pointer is given below

```
if forward at end of first half then begin
        reload second half;
        forward := forward + 1
end
else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
end
else forward := forward + 1;
```

Once the next lexeme is determined, *forward* is set to character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, *Lexeme Begin* is set to the character immediately after the lexeme just found.

2. Sentinels:

If we use the scheme of Buffer pairs we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch).

We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **EOF.** Look ahead code with sentinels is given below:

```
forward  := forward + 1;
if forward  = eof then begin
        if forward at end of first half then begin
                reload second half;
                 forward  := forward + 1
        end
else if forward at the second half then begin
        reload first half;
        move forward to beginning of first half
        end
else terminate lexical analysis
end;
```

4) Regular expression & regular definition

Regular Expression

1. 0 or 1
   0+1
2. 0 or 11 or 111
   0+11+111
3. Regular expression over ∑={a,b,c} that represent all string of length 3.
   (a+b+c)(a+b+c)(a+b+c)
4. String having zero or more a.
   a*
5. String having one or more a.
   a+
6. All binary string.
   (0+1)*
7. 0 or more occurrence of either a or b or both
   (a+b)*
8. 1 or more occurrence of either a or b or both
   (a+b)+
9. Binary no. end with 0
   (0+1)*0
10. Binary no. end with 1
    (0+1)*1
11. Binary no. starts and end with 1.

1(0+1)*1

12. String starts and ends with same character.

    0(0+1)*0 or a(a+b)*a

    1(0+1)*1 b(a+b)*b

13. All string of a and b starting with a

    a(a/b)*

14. String of 0 and 1 end with 00.

    (0+1)*00

15. String end with abb.

    (a+b)*abb

16. String start with 1 and end with 0.

    1(0+1)*0

17. All binary string with at least 3 characters and 3rd character should be zero.

    (0+1)(0+1)0(0+1)*

18. Language which consist of exactly Two b's over the set ∑={a,b,}

    a*ba*ba*

19. ∑={a,b} such that 3rd character from right end of the string is always a.

    (a/b)*a(a/b)(a/b)

20. Any no. of a followed by any no. of b followed by any no of c.

    a*b*c*

21. It should contain at least 3 one.

    (0+1)*1(0+1)*1(0+1)*1(0+1)*

22. String should contain exactly Two 1's

    0*10*10*

23. Length should be at least be 1 and at most 3.

    (0+1)+ (0+1) (0+1)+ (0+1) (0+1) (0+1)

24. No.of zero should be multiple of 3

    (1*01*01*01*)*+1*

25. ∑={a,b,c} where a are multiple of 3.

    ((b+c)*a (b+c)*a (b+c)*a (b+c)*)*

26. Even no. of 0.

    (1*01*01*)*

27. Odd no. of 1.

    0*(10*10*)*10*

28. String should have odd length.

    (0+1)((0+1)(0+1))*

29. String should have even length.

    ((0+1)(0+1))*

30. String start with 0 and has odd length.

    0((0+1)(0+1))*

31. String start with 1 and has even length.

1(0+1)((0+1)(0+1))*

32. Even no of 1
    (0*10*10*)*
33. String of length 6 or less
    $(0+1+\hat{})^6$
34. String ending with 1 and not contain 00.
    $(1+01)^+$
35. All string begins or ends with 00 or 11.
    (00+11)(0+1)*+(0+1)*(00+11)
36. All string not contains the substring 00.
    (1+01)* (^+0)
37. Every 0 is followed immediately by 1.
    1*(011*)*
38. Language of all string containing both 11 and 00 as substring.
    ((0+1)*00(0+1)*11(0+1)*)+ ((0+1)*11(0+1)*00(0+1)*)
39. Language of C identifier.
    (_+L)(_+L+D)*
40. The set of all string not congaing 101 as a substring.
    0*(1+000*)*0*
41. The language of all 0's and 1's having 1 at odd position.
    (1(0+1)*)(1+^)

Regular Definition

A regular definition gives names to certain regular expressions and uses those names in other regular expressions.

Here is a regular definition for the set of Pascal identifiers that is define as the set of strings of letter and digits beginning with a letters.

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

digit → 0 | 1 | 2 | . . . | 9

id → letter (letter | digit)*

The regular expression id is the pattern for the Pascal identifier token and defines letter and digit. Where letter is a regular expression for the set of all upper-case and lower case letters in the alphabet and digit is the regular for the set of all decimal digits.

5) Explain Finite autometa (NFA & DFA)

- It is a mathematical model- state transition diagram
- It is a Recognizer for a given language
- 5-tuple {Q, Σ , δ, q_0, F}
  - Q is a finite set of states
  - Σ is a finite set of input
  - *f* transition function Q x Σ

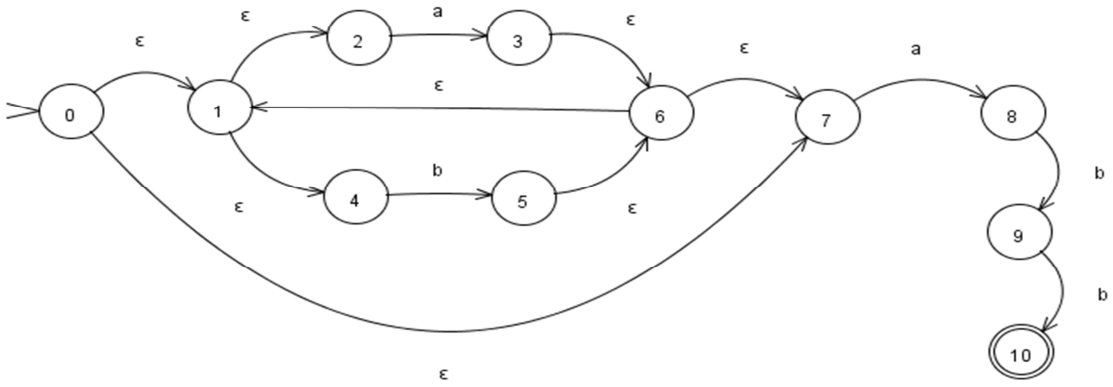
- q_0, F initial and final state repsectively

There are mainly two types of finite autometa, Non deterministic autometa and Deterministic autometa. Difference between NFA and DFA.

Parameter	NFA	FA
Transition	Non deterministic	Deterministic
No. of states	NFA has a fewer number of states.	More, if NFA contains Q states then the corresponding FA will have $<=2^Q$ states.
Power	NFA is as powerful as DFA	FA is powerful as an NFA
Design	Easy to design due to non-determinism	More difficult to design
Acceptance	It is difficult to find whether w € L as there are several paths. Backtracking is required to explore several parallel paths.	It is easy to find whether w € L as transition are deterministic.

6) Conversion from NFA to DFA using Thompson's rule

Example: (a/b)*abb



- **ε –** closure (0) = {0,1,2,4,7} ---- Let A
- Move(A,a) = {3,8}

ε – closure (Move(A,a)) = {1,2,3,4,6,7,8}---- Let B
Move(A,b) = {5}
ε – closure (Move(A,b)) = {1,2,4,5,6,7}---- Let C

- Move(B,a) = {3,8}

  ε – closure (Move(B,a)) = {1,2,3,4,6,7,8}---- B
  Move(B,b) = {5,9}
  ε – closure (Move(B,b)) = {1,2,4,5,6,7,9}---- Let D

- Move(C,a) = {3,8}

  ε – closure (Move(C,a)) = {1,2,3,4,6,7,8}---- B
  Move(C,b) = {5}
  ε – closure (Move(C,b)) = {1,2,4,5,6,7}---- C

- Move(D,a) = {3,8}

  ε – closure (Move(D,a)) = {1,2,3,4,6,7,8}---- B
  Move(D,b) = {5,10}
  ε – closure (Move(D,b)) = {1,2,4,5,6,7,10}---- Let E

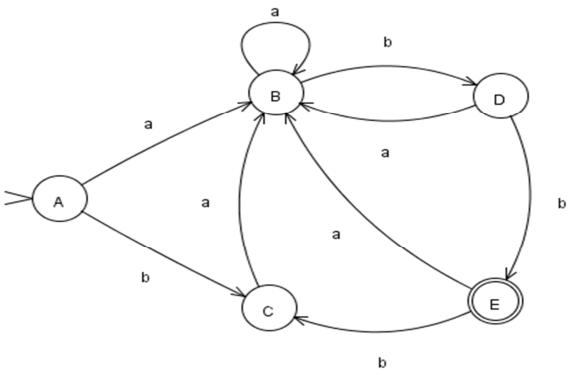
- Move(E,a) = {3,8}

  ε – closure (Move(E,a)) = {1,2,3,4,6,7,8}---- B
  Move(E,b) = {5}
  ε – closure (Move(E,b)) = {1,2,4,5,6,7}---- C

Transition Table:

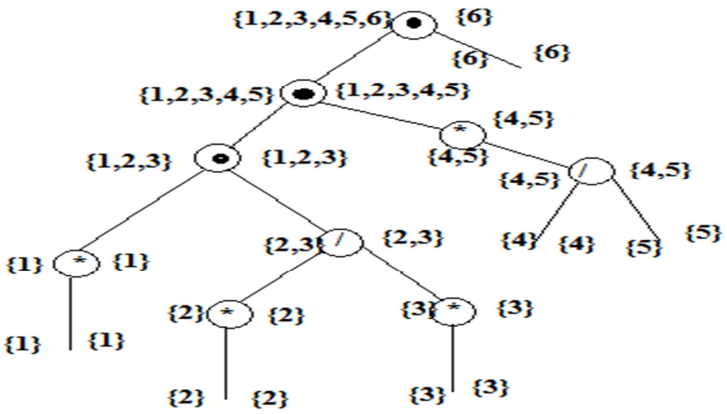
	a	**b**
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

DFA:



7) Conversion from Regular Expression to DFA without constructing NFA

Example: a*(b*/ c*) (a / c) *



- For root node

i = lastpos(C_1) = {1,2,34,5,}

followpos(i) = firstpos(C_2) = {6}

followpos(1) = {6}

followpos(2) = {6}

followpos(3) = {6}

followpos(4) = {6}

followpos(5) = {6}

i = lastpos(C_1) = {1,2,3}

followpos(i) = firstpos(C_2) = {4,5}

followpos(1) = {4,5}

followpos(2) = {4,5}

followpos(3) = {4,5}

i = lastpos(n) = {4,5}

followpos(i) = firstpos(n) = {4,5}

followpos(4) = {4,5}

followpos(5) = {4,5}

i = lastpos(C_1) = {1}

followpos(i) = firstpos(C_2) = {2,3}

followpos(1) = {2,3}

i = lastpos(n) = {2}

followpos(i) = firstpos(n) = {2}

followpos(2) = {2}

i = lastpos(n) = {3}

followpos(i) = firstpos(n) = {3}

followpos(3) = {3}

i	Followpos(i)
1	{1,2,3,4,5,6}
2	{2,4,5,6}
3	{3,4,5,6}
4	{4,5,6}
5	{4,5,6}

Construct DFA:

Initial node = firstpos (root node)

= {1,2,3,4,5,6}

δ ((1,2,3,4,5,6) , a) = followpos(1) U followpos(4)

={1,2,3,4,5,6}

δ ((1,2,3,4,5,6) , b) = followpos(2)

={2,4,5,6}

δ ((1,2,3,4,5,6) , c) = followpos(3) U followpos(5)

={3,4,5,6}

δ ((2 ,4,5,6) , a) = followpos(4)

={4,5,6}

δ ((2 ,4,5,6) , b) = followpos(2)

={2,4,5,6}

δ ((2 ,4,5,6) , c) = followpos(5)

={4,5,6}

δ ((3,4,5,6) , a) = followpos(4)

={4,5,6}

δ ((3,4,5,6) , b) = Φ

δ ((3,4,5,6) , c) = followpos(3) U followpos(5)

={3,4,5,6}

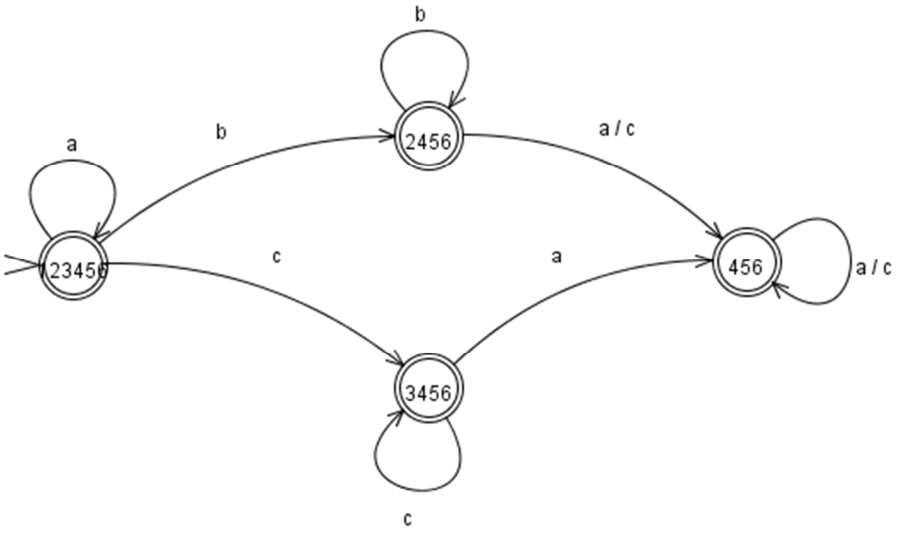
δ ((4,5,6) , a) = followpos(4)

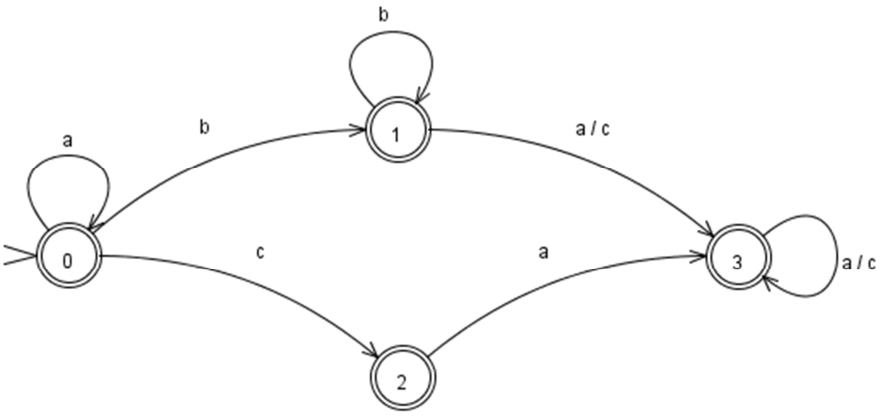
={4,5,6}

δ ((4,5,6) , b) = Φ

δ ((4,5,6) , c) = followpos(5)

={4,5,6}


b
a
b
2456
a / c
2345
c
a
456
a / c
3456
c


b
a
b
1
a / c
0
c
a
3
a / c
2