NPTEL ONLINE CERTIFICATION COURSES

Compiler Design

Lexical Analysis

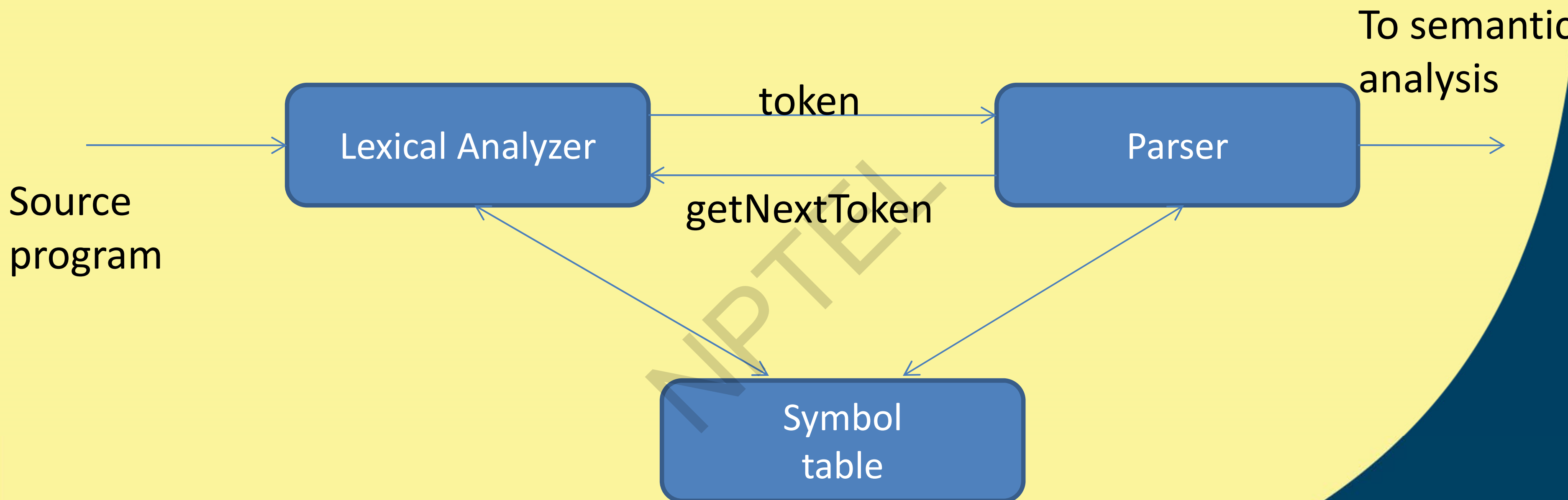
Santanu Chattopadhyay

Electronics and Electrical Communication Engineering

CONCEPTS COVERED

- Role of Lexical Analyzer
- Tokens, Patterns, Lexemes
- Lexical Errors and Recovery
- Specification of Tokens
- Recognition of Tokens
- Finite Automata
- NFA and DFA
- Tool lex
- Conclusion

Role of lexical analyzer



Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

Tokens, Patterns and Lexemes

- A token is a pair – a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

Example

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but “ surrounded by “	“core dumped”

Attributes for tokens

- E = M * C ** 2
  - <id, pointer to symbol table entry for E>
  - <assign-op>
  - <id, pointer to symbol table entry for M>
  - <mult-op>
  - <id, pointer to symbol table entry for C>
  - <exp-op>
  - <number, integer value 2>

Lexical errors

- Some errors are out of power of lexical analyzer to recognize:
  - fi (a == f(x)) …
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
  - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely

											E	=	M	*	C	*	*	2	eof							

Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
    - letter(letter | digit)*
- Each regular expression is a pattern specifying the form of strings

Regular Expressions

- Ɛ is a regular expression denoting the language L(Ɛ) = {Ɛ}, containing only the empty string
- If *a* is a symbol in Σ then *a* is a regular expression, L(*a*) = {*a*}
- If *r* and *s* are two regular expressions with languages L(*r*) and L(*s*), then
  - *r*|*s* is a regular expression denoting the language L(*r*) $\cup$ L(*s*), containing all strings of L(r) and L(s)
  - *rs* is a regular expression denoting the language L(*r*)L(*s*), created by concatenating the strings of L(*s*) to L(*r*)
  - *r** is a regular expression denoting (L(*r*))*, the set containing zero or more occurrences of the strings of L(*r*)
  - (*r*) is a regular expression corresponding to the language L(*r*)

Regular definitions

d1 -> r1

d2 -> r2

…

dn -> rn

- Example:

  letter_ -> A | B | … | Z | a | b | … | Z | _

  digit -> 0 | 1 | … | 9

  id -> letter_ (letter_ | digit)*

Extensions

- One or more instances: (r)+
- Zero of one instances: r?
- Character classes: [abc]

- Example:
  – letter_ -> [A-Za-z_]
  – digit -> [0-9]
  – id -> letter_(letter_|digit)*

Examples with $\Sigma = \{0, 1\}$

- (0|1)*: All binary strings including the empty string
- (0|1)(0|1)*: All nonempty binary strings
- 0(0|1)*0: All binary strings of length at least 2, starting and ending with 0s
- (0|1)*0(0|1)(0|1)(0|1): All binary strings with at least three characters in which the third-last character is always 0
- 0*10*10*10*: All binary strings possessing exactly three 1s

Example

Set of floating-point numbers:

(+|-|Ɛ) digit (digit)*(. digit (digit)*|Ɛ)((E(+|-|Ɛ) digit (digit)*)|Ɛ)

Recognition of tokens

- Starting point is the language grammar to understand the tokens:

stmt -> **if** expr **then** stmt
    | **if** expr **then** stmt **else** stmt
    | ε
expr -> term **relop** term
    | term
term -> **id**
    | **number**

Recognition of tokens (cont.)

- The next step is to formalize the patterns:

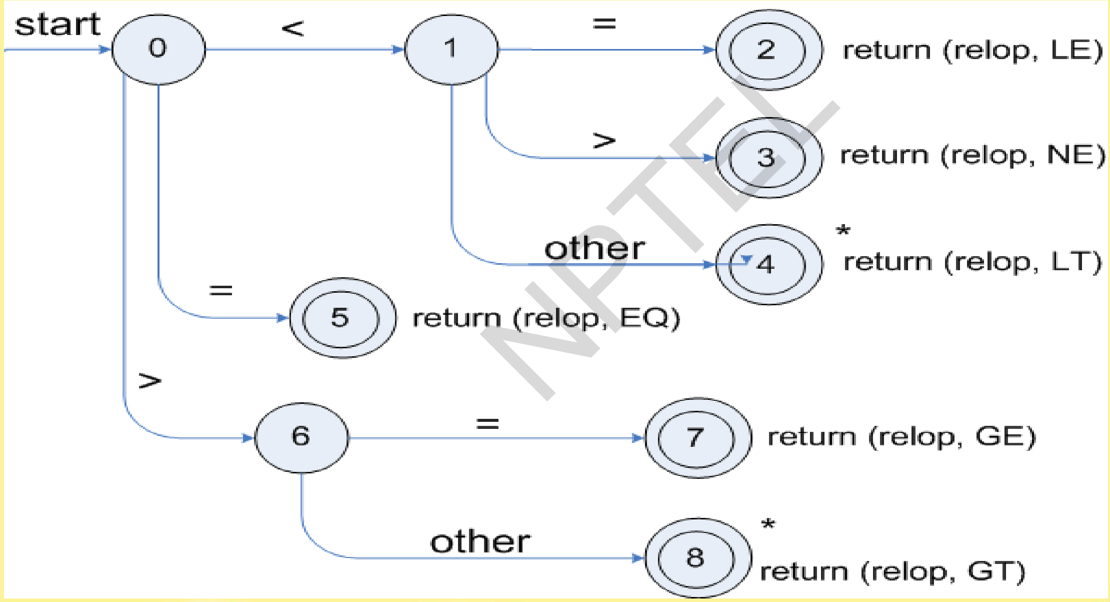
  digit -> [0-9]
  Digits -> digit+
  number -> digit(.digits)? (E[+-]? Digit)?
  letter -> [A-Za-z_]
  id -> letter (letter|digit)*
  If -> if
  Then -> then
  Else -> else
  Relop -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

  ws -> (blank | tab | newline)+

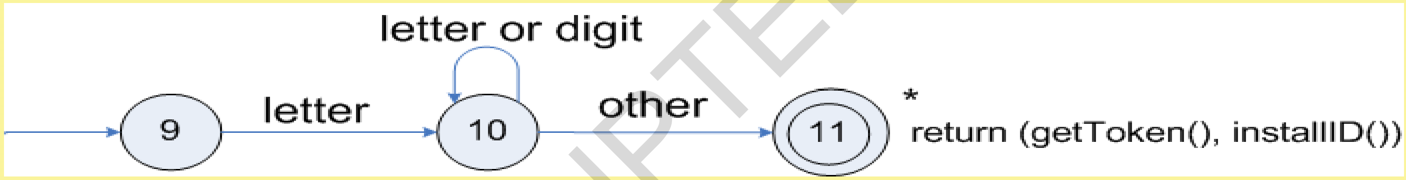
Transition diagrams

- Transition diagram for relop



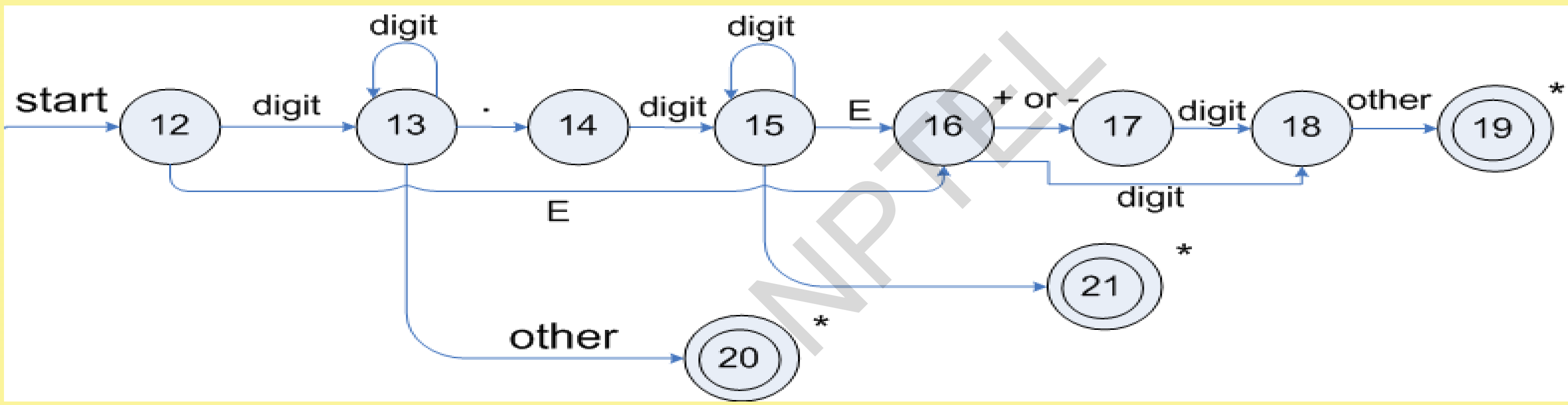
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



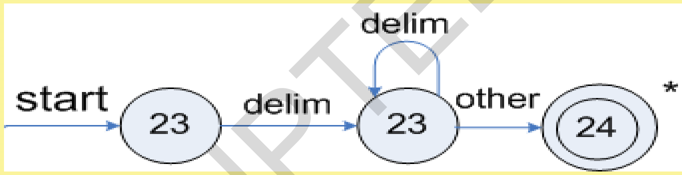
Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Transition diagrams (cont.)

- Transition diagram for whitespace



Architecture of a transition-diagram-based lexical analyzer

```
TOKEN getRelop()
{
        TOKEN retToken = new (RELOP)
        while (1) { /* repeat character processing until a
                                        return or failure occurs          */
        switch(state) {
                case 0: c= nextchar();
                                if (c == '<') state = 1;
                                else if (c == '=') state = 5;
                                else if (c == '>') state = 6;
                                else fail();      /* lexeme is not a relop */
                                break;
                case 1: …
                …
                case 8: retract();
                                retToken.attribute = GT;
                                return(retToken);
        }
```

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions state $\xrightarrow{\text{input}}$ state

Finite Automata

- Transition

$$s_1 \xrightarrow{a} s_2$$

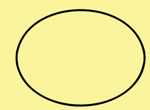
- Is read

In state s_1 on input “a” go to state s_2

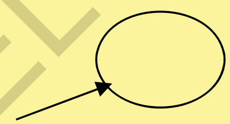
- If end of input
  - If in accepting state => accept, othewise => reject
- If no transition possible => reject

Finite Automata State Graphs

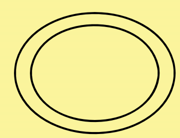
- A state



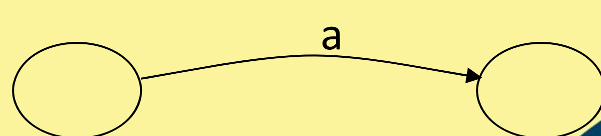
- The start state



- An accepting state

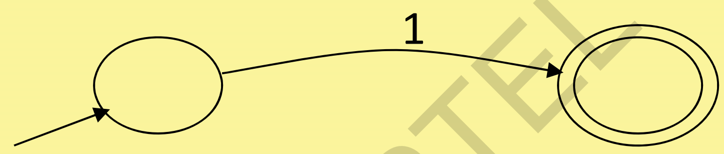


- A transition



A Simple Example

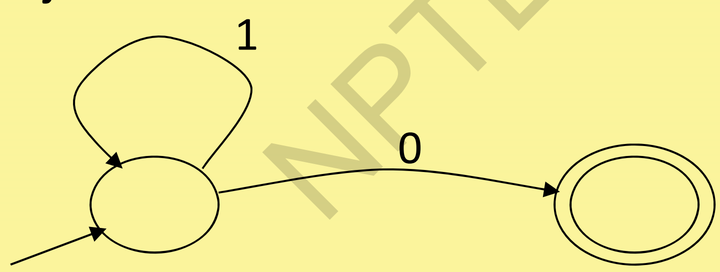
- A finite automaton that accepts only “1”



- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

Another Simple Example

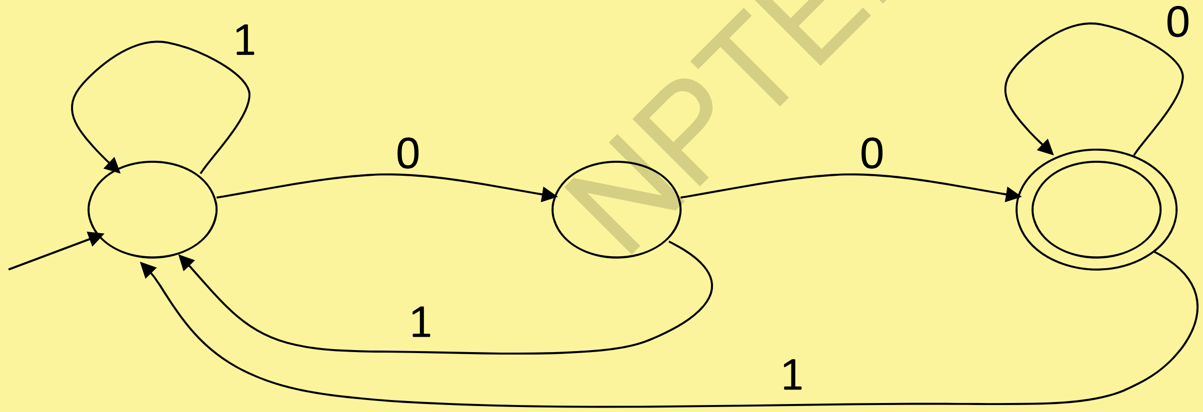
- A finite automaton accepting any number of 1’s followed by a single 0
- Alphabet: {0,1}



- Check that “1110” is accepted but “110…” is not

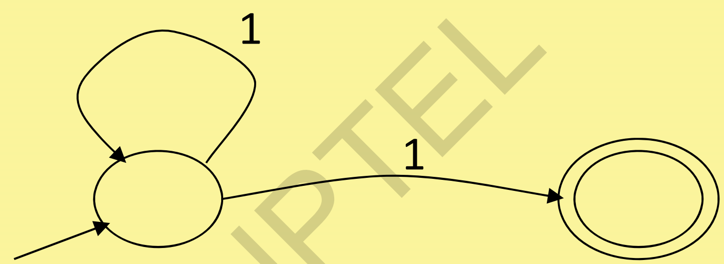
And Another Example

- Alphabet {0,1}
- What language does this recognize?



And Another Example

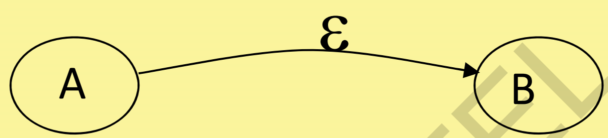
- Alphabet still { 0, 1 }



- The operation of the automaton is not completely defined by the input
  - On input “11” the automaton could be in either state

Epsilon Moves

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input

Deterministic and Nondeterministic Automata

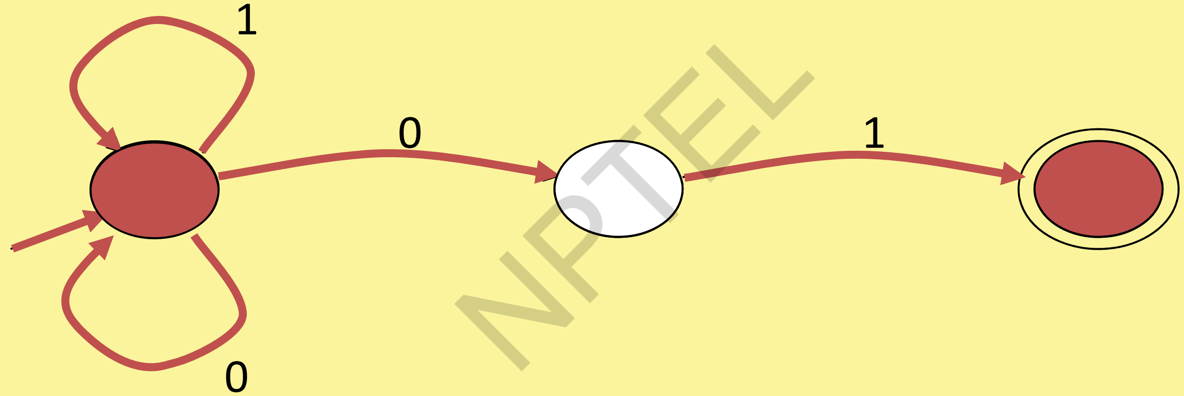
- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- *Finite* automata have *finite* memory
  - Need only to encode the current state

Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 1
- Rule: NFA accepts if it can get in a final state

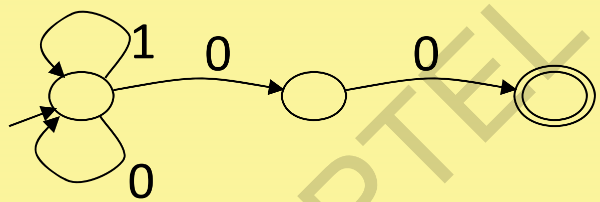
NFA vs. DFA (1)

- NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are easier to implement
  - There are no choices to consider

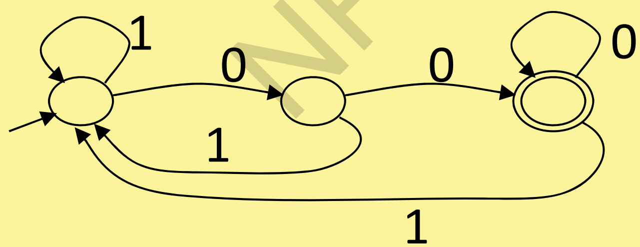
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA



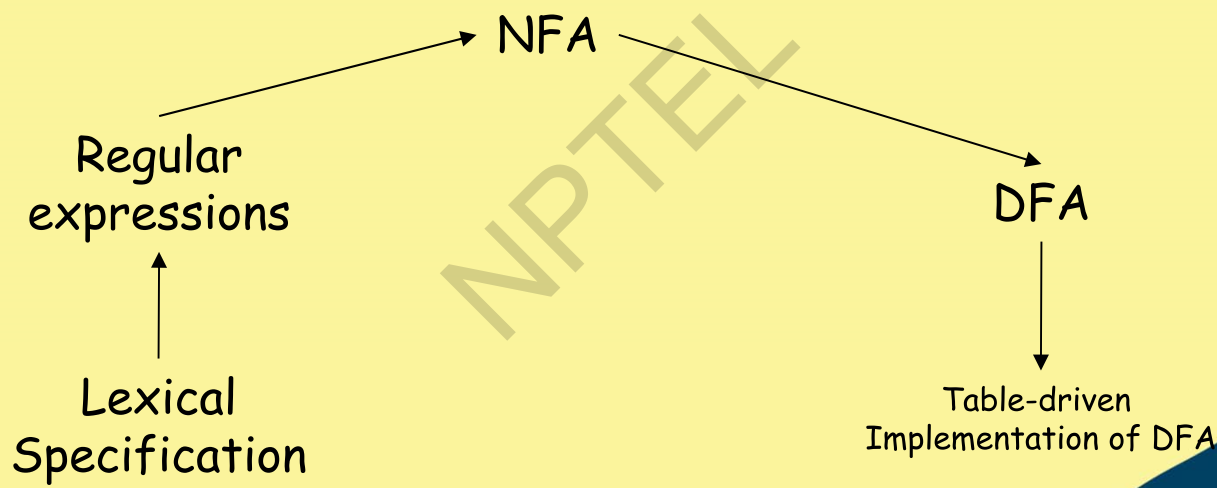
DFA



- DFA can be exponentially larger than NFA

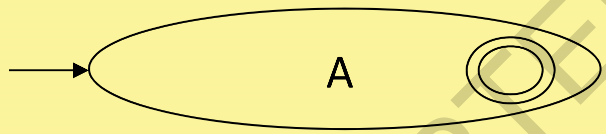
Regular Expressions to Finite Automata

- High-level sketch

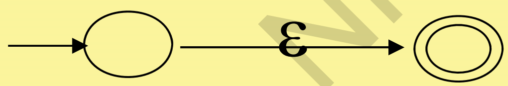


Regular Expressions to NFA (1)

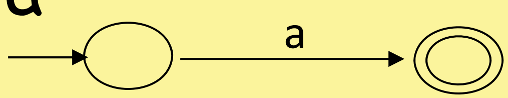
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A



- For ε

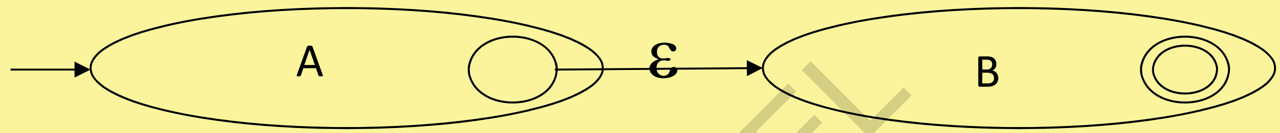


- For input a

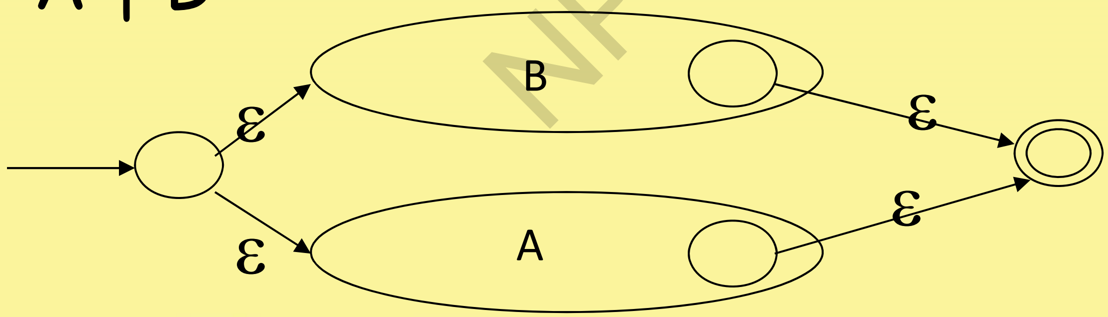


Regular Expressions to NFA (2)

- For AB

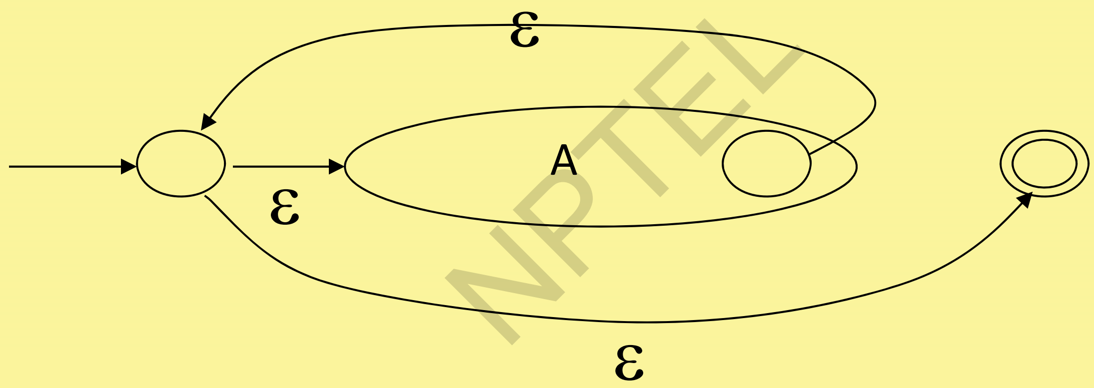


- For A | B



Regular Expressions to NFA (3)

- For A*

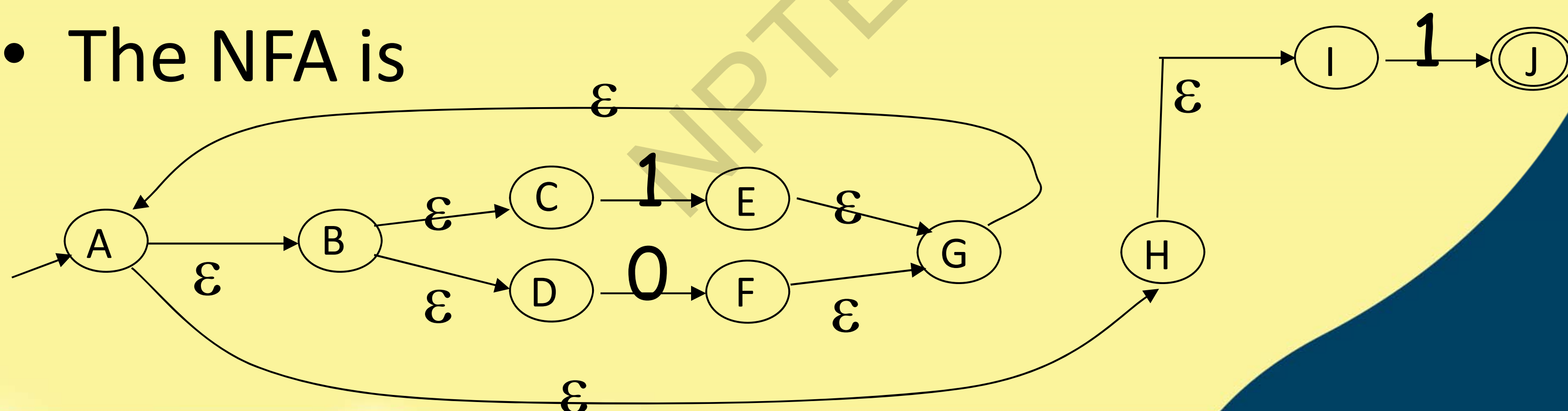


Example of RegExp -> NFA conversion

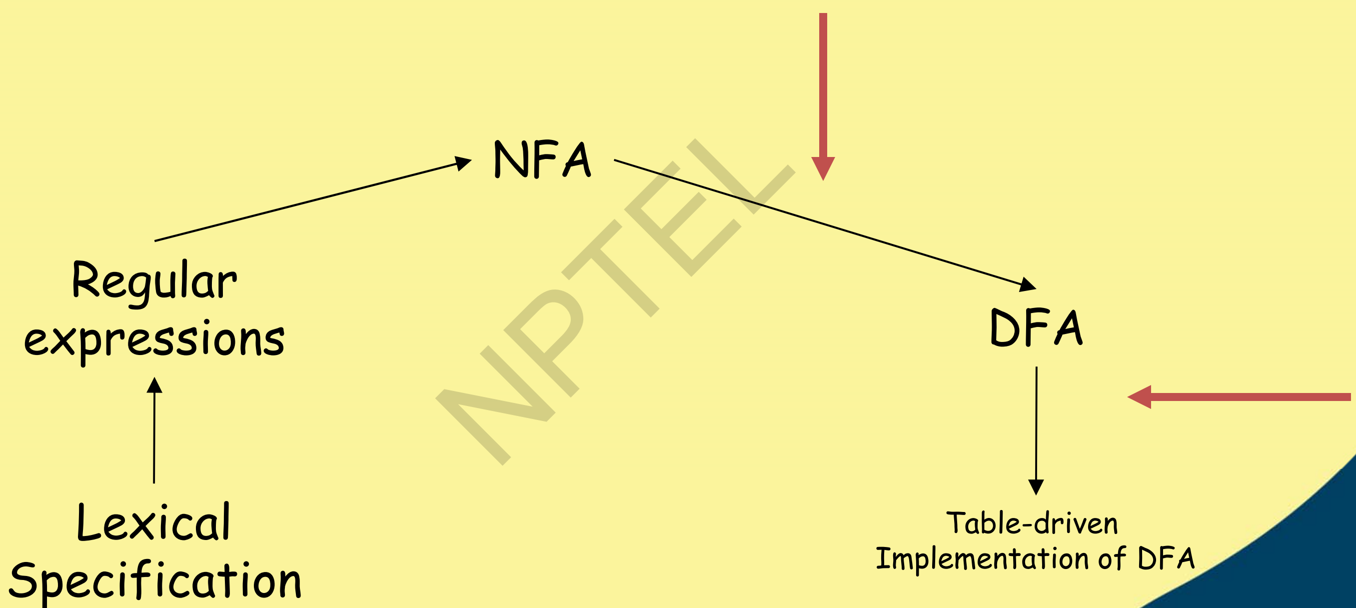
- Consider the regular expression

$$(1 \mid 0)^*1$$

- The NFA is



Next



NFA
Regular expressions
DFA
Lexical Specification
Table-driven Implementation of DFA

NFA to DFA. The Trick

- Simulate the NFA
- Each state of resulting DFA

  = a non-empty subset of states of the NFA
- Start state

  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^{a} S'$ to DFA iff
  - S' is the set of NFA states reachable from the states in S after seeing the input a
    - considering ε-moves as well