Systems Software

Lexical Analysis

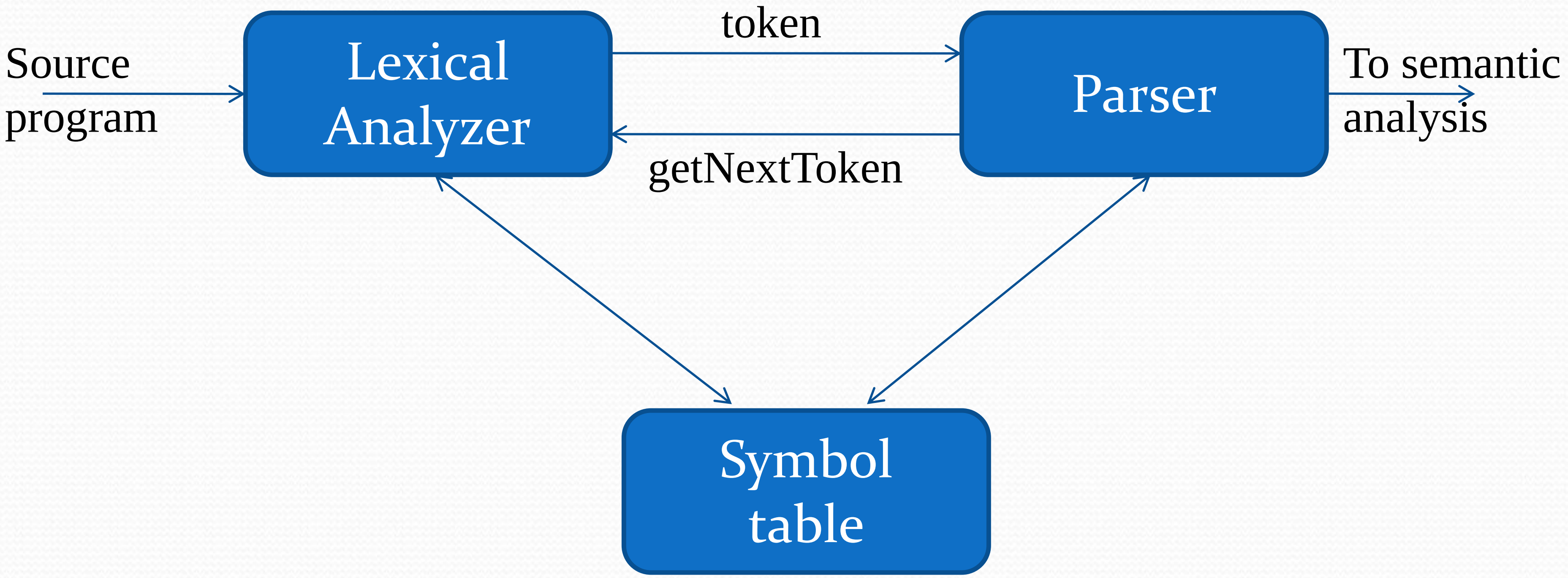
Example

- Real:= position, rate, initial
- Position=initial + rate *60
- Translate the above statement using phases of compiler

Outline

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Lexical analyzer generator
- Finite automata
- Design of lexical analyzer generator

The role of lexical analyzer



The role of lexical analyzer

- 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.

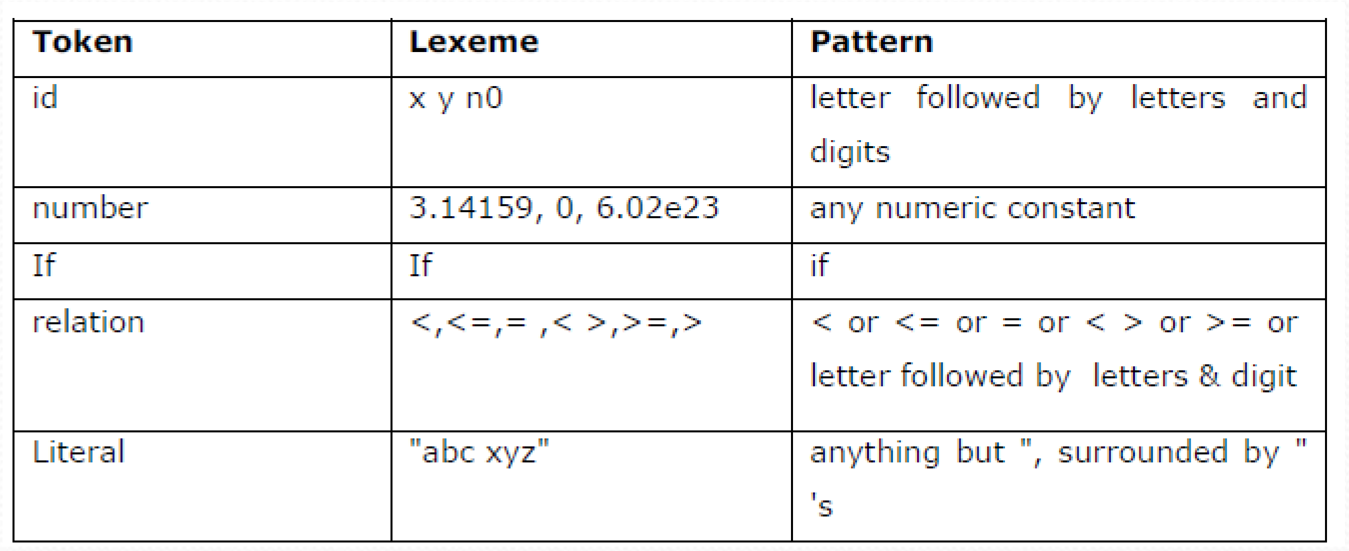
- Some lexical analyzer are divided into cascade of two phases, the first called scanning and second is “lexical analysis”.
- The scanner is responsible for doing simple task while lexical analysis does the more complex task.

- 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.

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, Pattern and Lexemes…



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

if(x<=5)

- Token – if (keyword),
- X (id),
- <= (relation),
- 5 (number)

- Lexeme - if , x ,<=, 5

total = sum + 12.5

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

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

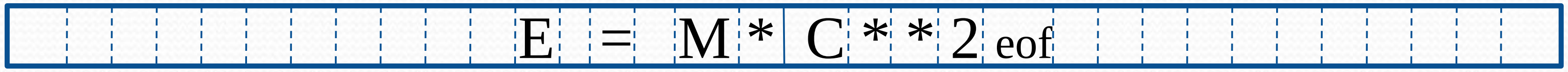
- Some errors are out of power of lexical analyzer to recognize:
  - fi (a == f(x)) …
- Scenario 1
- If the string fi encounters in C program for the first time in context.
- Scenario 2
- What if lexical analyzer unable to proceed because of no match of pattern.

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

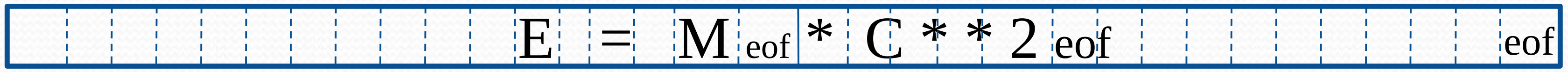
Input buffering

- 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



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

Sentinels



```
forward =forward+1
If forward= eof then begin
        if  forward is at end of first half then begin
                reload second half;
                forward = forward+1;
        end
        else if  forward is at end of second half then begin
                reload first half;\
                move forward to beginning of first half;
        end
        else //eof within a buffer signifying end of input

                terminate lexical analysis
end
```

Specification of tokens

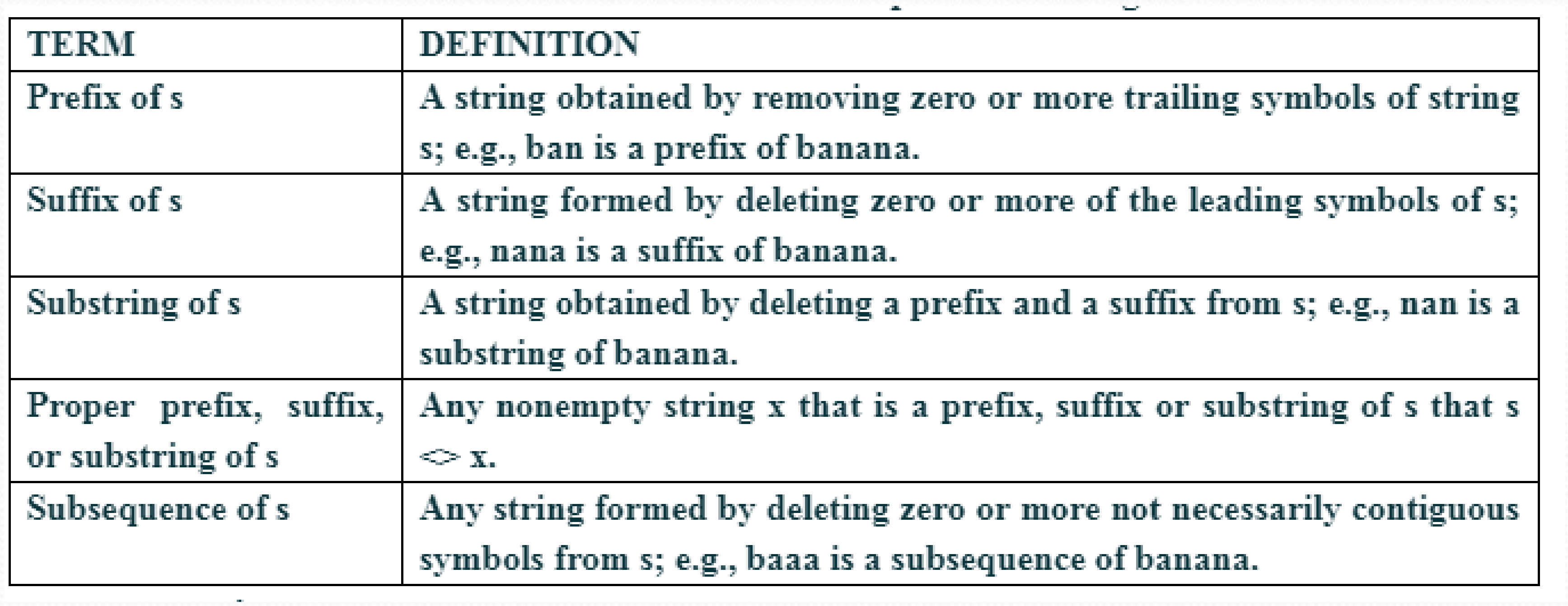
- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
    - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

Specification of tokens

- **Strings and Languages:**
- The term alphabet or character class denotes any finite set of symbols.
- Examples of symbols are letters and characters.
- e.g., set {0,1} is the binary alphabet.
- The term sentence and word are often used as synonyms for the term string.
- The length of a string s is written as |s| - is the number of occurrences of symbols in s.
- e.g., string “banana” is of length six.

Specification of tokens

- The empty string denoted by ε – length of empty string is zero.
- The term language denotes any set of strings over some fixed alphabet.
- e.g., {ε} – set containing only empty string is language under φ.
- If x and y are strings, then the concatenation of x and y (written as xy) is the string formed by appending y to x. x = dog and y = house; then xy is doghouse.



TERM	DEFINITION
Prefix of s	A string obtained by removing zero or more trailing symbols of string s; e.g., ban is a prefix of banana.
Suffix of s	A string formed by deleting zero or more of the leading symbols of s; e.g., nana is a suffix of banana.
Substring of s	A string obtained by deleting a prefix and a suffix from s; e.g., nan is a substring of banana.
Proper prefix, suffix, or substring of s	Any nonempty string x that is a prefix, suffix or substring of s that s <> x.
Subsequence of s	Any string formed by deleting zero or more not necessarily contiguous symbols from s; e.g., baaa is a subsequence of banana.

Terms for parts of a string

Operations on Languages

- **There are several operations that can be applied to languages:**
- **Let *L*** be the set $\{A, B, \ldots, Z, a, b, \ldots, z\}$
- Let D be the set $\{0,1, \ldots, 9\}$
- L is alphabet consist of upper and lower case letters.
- D is the alphabet set of 10 digits.

Operations on Languages

- Some examples of new languages created from L and D by applying some operations
- $\boldsymbol{L} \cup \boldsymbol{D}$
- $\boldsymbol{LD}$
- $\boldsymbol{L}^4$
- $\boldsymbol{L}^*$
- $\boldsymbol{L}(\boldsymbol{L} \cup \boldsymbol{D})$
- $\boldsymbol{D}^+$

OPERATION	DEFINITION
Union of L and M. written L υ M	L υ M = { s | s is in L or s is in M }
Concatenation of L and M. written LM	LM = { st | s is in L and t is in M }
Kleene closure of L. written L*	L* denotes “zero or more concatenation of” L.
Positive closure of L. written L+	L^+ denotes “one or more Concatenation of” L.

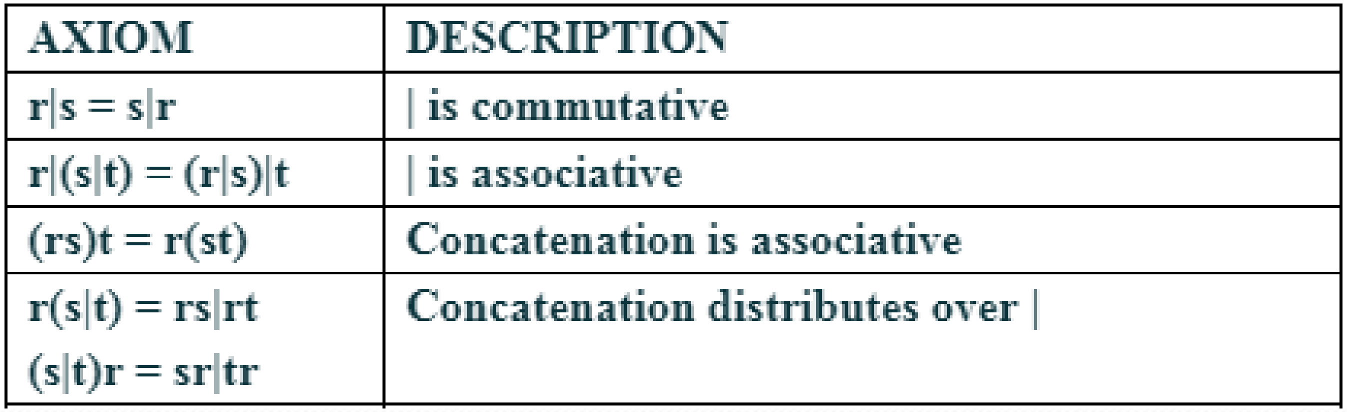
Regular Expression

- It allows defining the sets to form tokens precisely.
- e.g., letter (letter | digit) *
- Defines a Pascal identifier – which says that the identifier is formed by a letter followed by zero or more letters or digits.
- A regular expression is built up out of simpler regular expressions using a set of defining rules.
- Each regular expression r denotes a language L(r).

Regular expressions

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) ∪ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denting L(r)

- The regular expression $(a|b)(a|b)$ denotes which language?
- The regular expression $a|a^*b$ denotes which language?



AXIOM	DESCRIPTION
r|s = s|r	| is commutative
r|(s|t) = (r|s)|t	| is associative
(rs)t = r(st)	Concatenation is associative
r(s|t) = rs|rt (s|t)r = sr|tr	Concatenation distributes over |

Algebraic Properties of regular expressions

Regular definitions

We may wish to give names to regular expression and to define regular expressions using these names as if they were symbols.

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

d1 -> r1

d2 -> r2

…

dn -> rn

Where each d_i is a distinct name, and each r_i is a regular expression

- 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)*

- Unsigned numbers in pascal are strings such as,5280,39.37,1243.25E+2,6.33E5

- Unsigned numbers in pascal are strings such as 5280,39.37,6.336E4 or 1.894E-4

- $digit \rightarrow 0|1| \ldots |9$
- $digits \rightarrow digit\ digit^*$
- $fraction \rightarrow .digits \mid \in$
- $Exp_{fraction} \rightarrow (E(+|-|\in)digits)| \in$
- $num \rightarrow digits\ fraction\ Exp_{fraction}$

- Notational shorthands
- $digit \ \rightarrow 0|1| \dots |9$
- $digits \ \rightarrow digit^{+}$
- $fraction \ \rightarrow (.digits\ )?$
- $Exp_{fraction} \rightarrow (E(+|-)?\,digits)?$
- $num \ \rightarrow digits\ fraction\ Exp_{fraction}$

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:

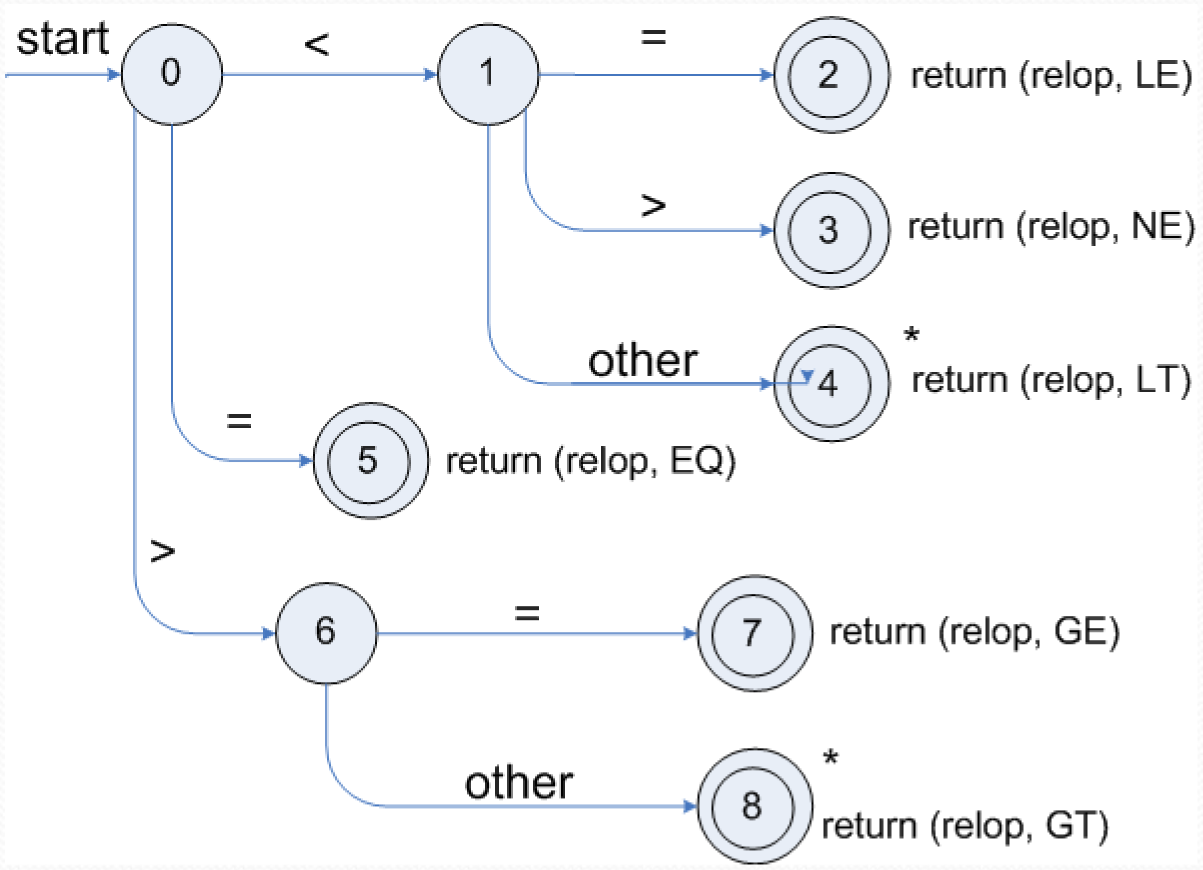
  ws -> (blank | tab | newline)+

Transition Diagram

- It is intermediate steps in the construction of a lexical analyzer
- It depicts the actions that take place when a lexical analyzer is called by parser to get next token.
- consider input buffer with lexeme_begin points to the character.
- Transition diagram is used to keep the information about characters that seen as forward.

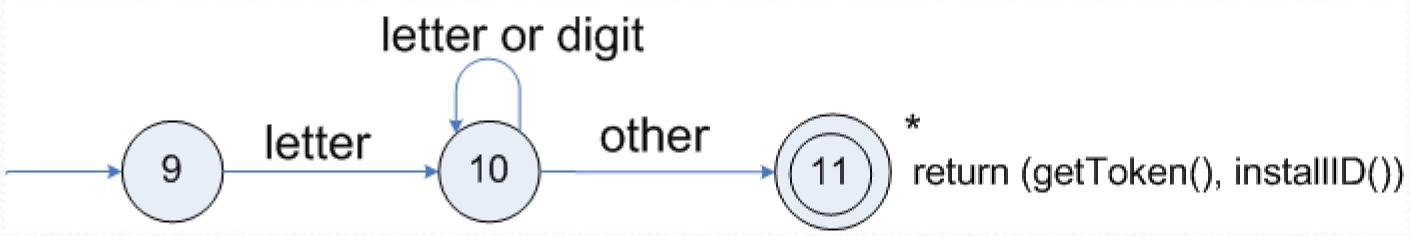
Transition diagrams

- Transition diagram for relop



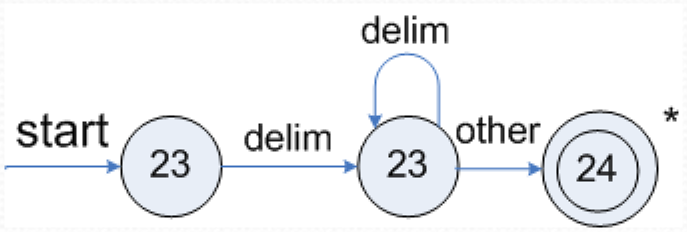
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers

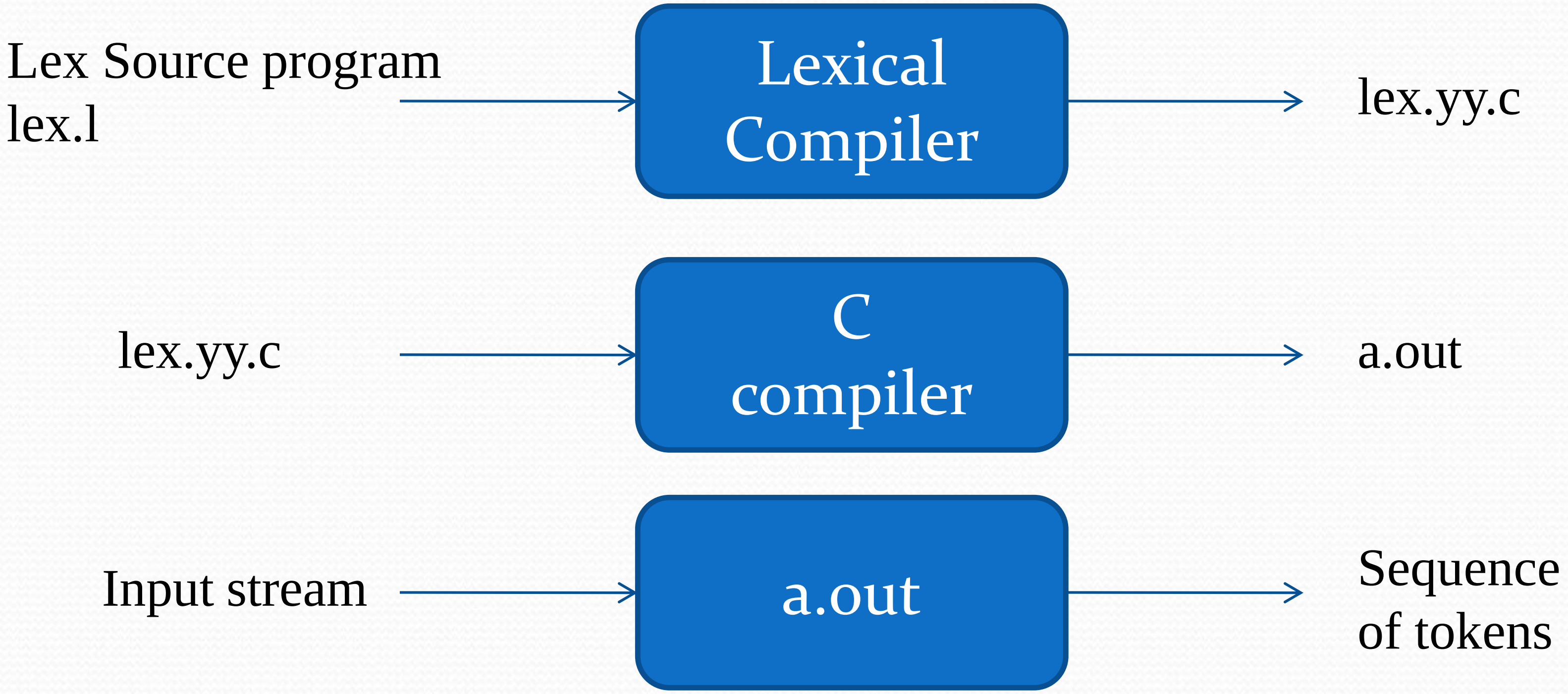


Transition diagrams (cont.)

- Transition diagram for whitespace



Lexical Analyzer Generator - Lex



Structure of Lex programs

declarations
%%
translation rules $\longrightarrow$ Pattern {Action}
%%
auxiliary functions

Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID); }
{number}    {yylval = (int) installNum(); return(NUMBER);}
…
```

```
Int installID() {/* funtion to install the
    lexeme, whose first character is
    pointed to by yytext, and whose
    length is yyleng, into the symbol
    table and return a pointer thereto
    */
}

Int installNum() { /* similar to
    installID, but puts numerical
    constants into a separate table */
}
```

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 $\rightarrow^{\text{input}}$ state

Finite Automata

- Transition

$$s_1 \rightarrow^{a} s_2$$

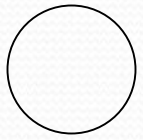
- Is read

In state s_1 on input "a" go to state s_2

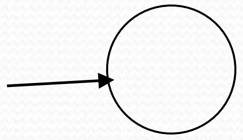
- If end of input
  - If in accepting state => accept, othewise => reject
- If no transition possible => reject

Finite Automata State Graphs

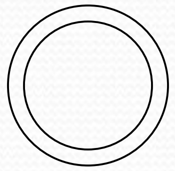
- A state



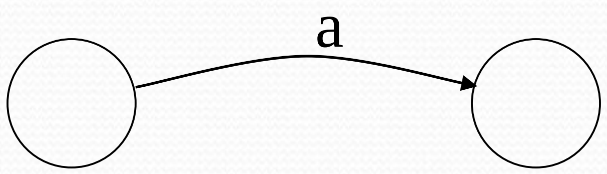
- The start state



- An accepting state

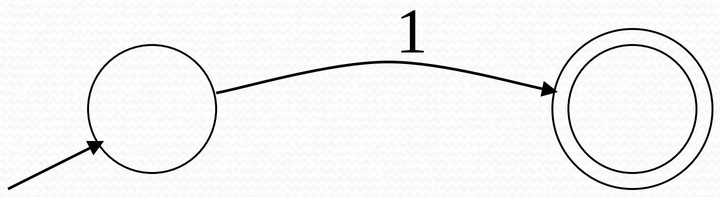


- A transition



A Simple Example

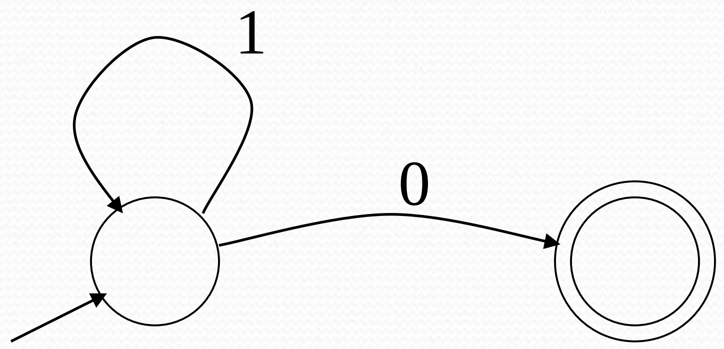
- A finite automaton that accepts only “1”



- 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

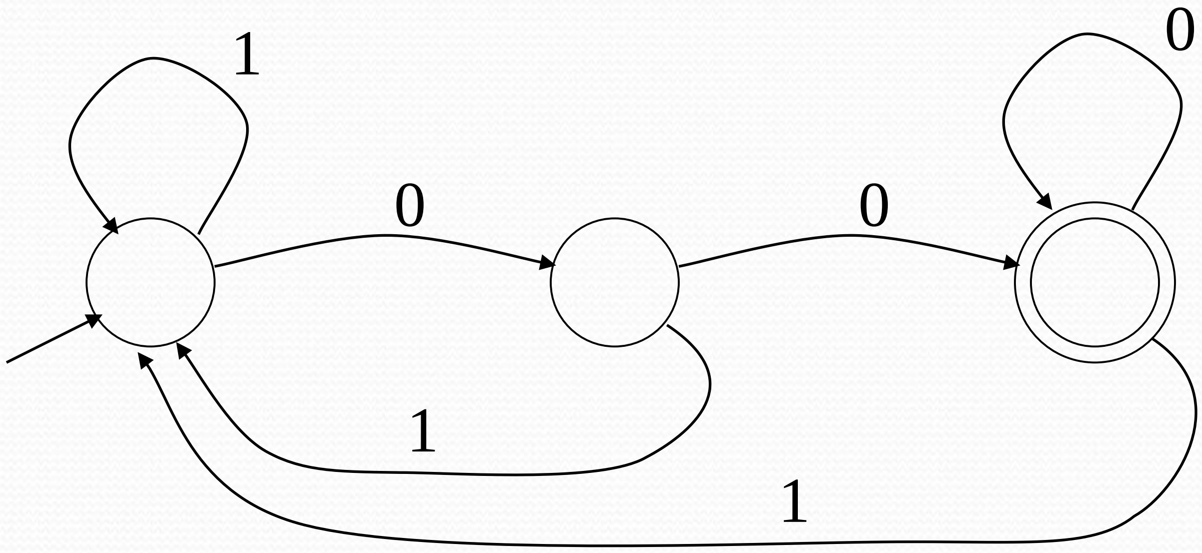
- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



- Check that "1110" is accepted but "110…" is not

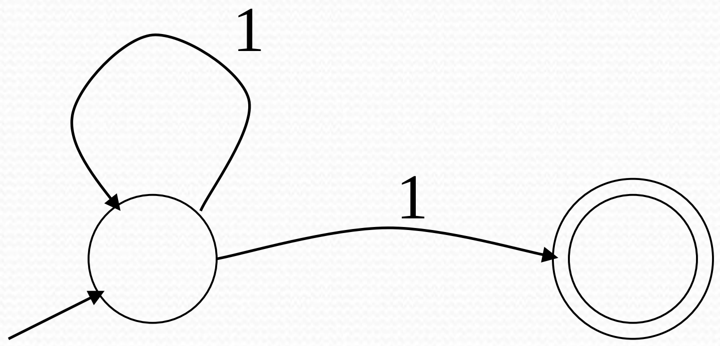
And Another Example

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



And Another Example

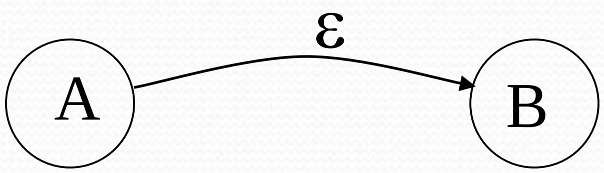
- Alphabet still { 0, 1 }



- 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


$$\begin{tikzcd} A \arrow[r, bend left, "\varepsilon"] & B \end{tikzcd}$$

- 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

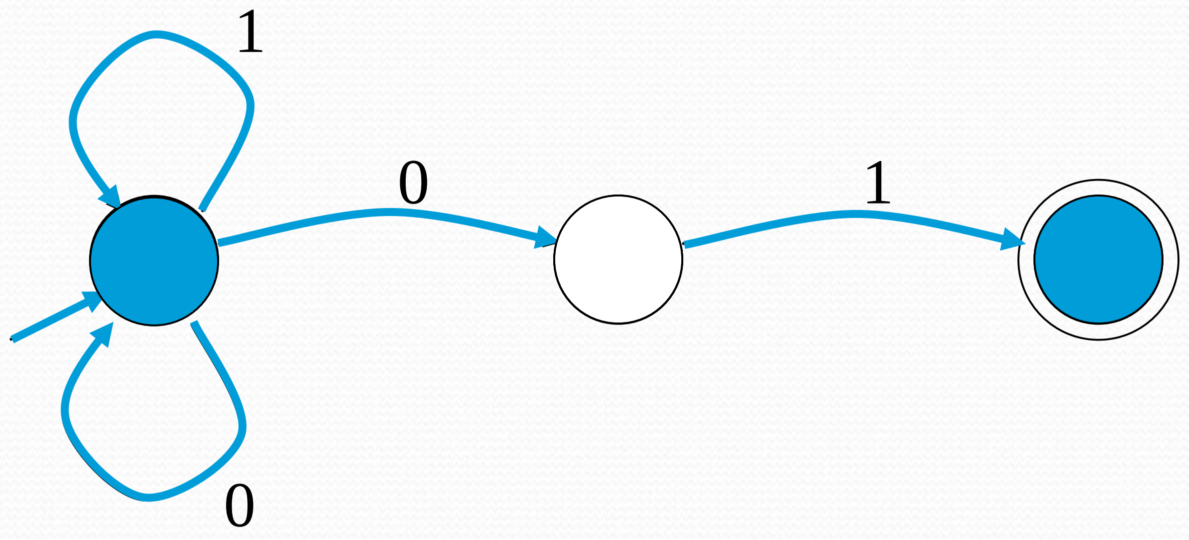
Execution of Finite Automata

- 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 <u>can</u> get in a final state

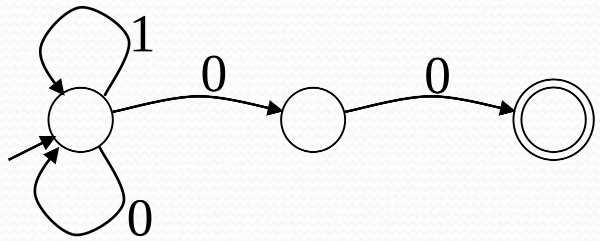
NFA vs. DFA (1)

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

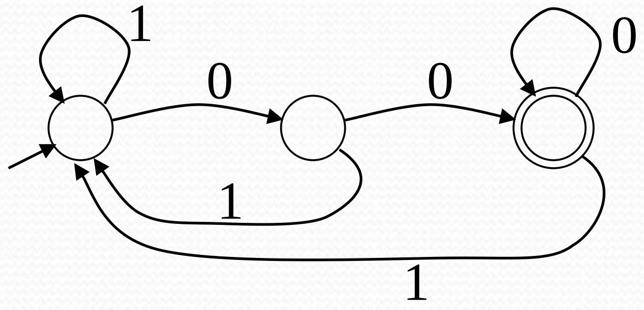
NFA vs. DFA (2)

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

NFA



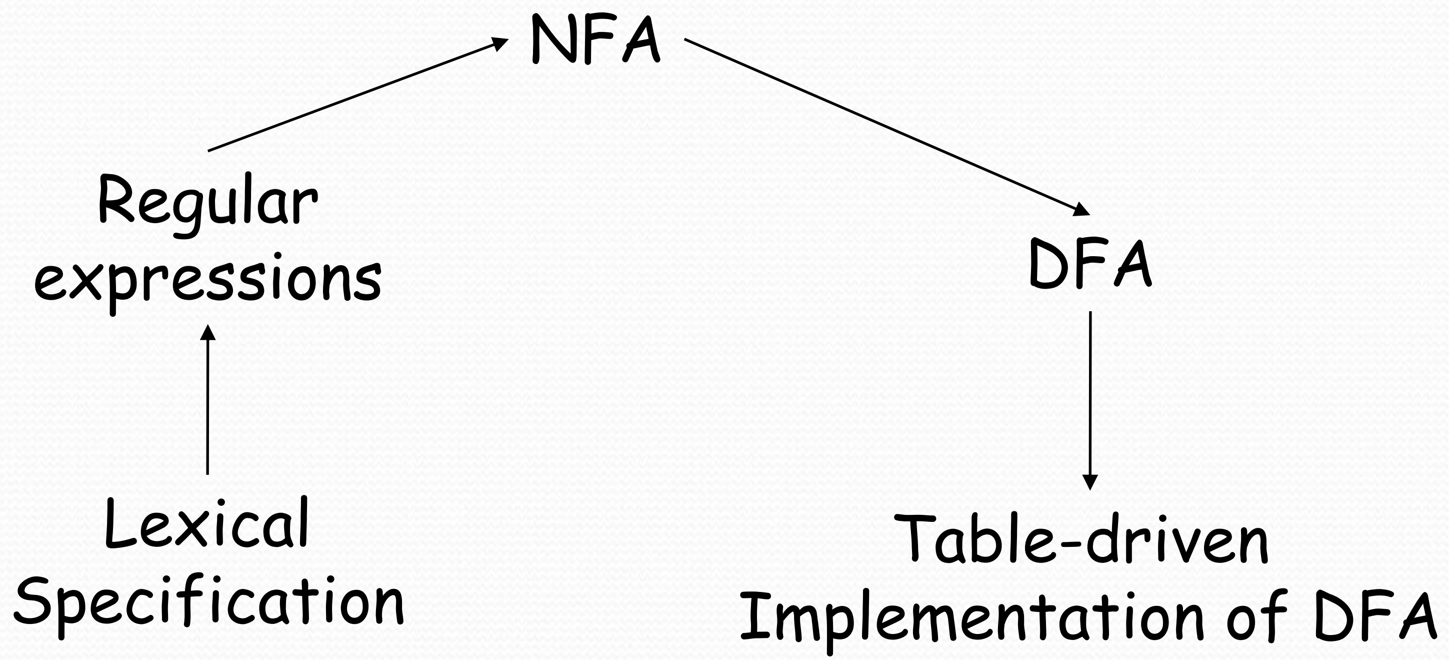
DFA



- DFA can be exponentially larger than NFA

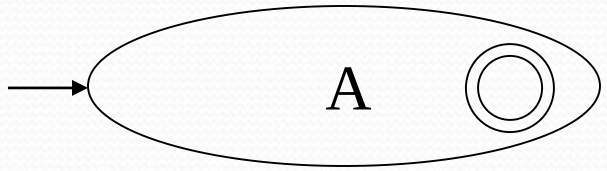
Regular Expressions to Finite Automata

- High-level sketch

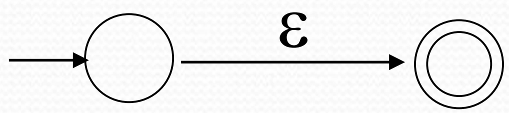


Regular Expressions to NFA (1)

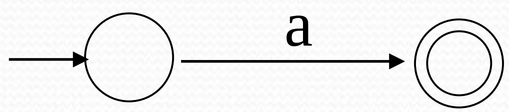
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A



- For ε

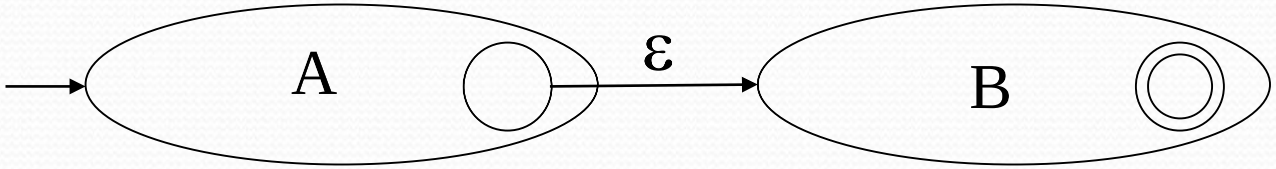


- For input a

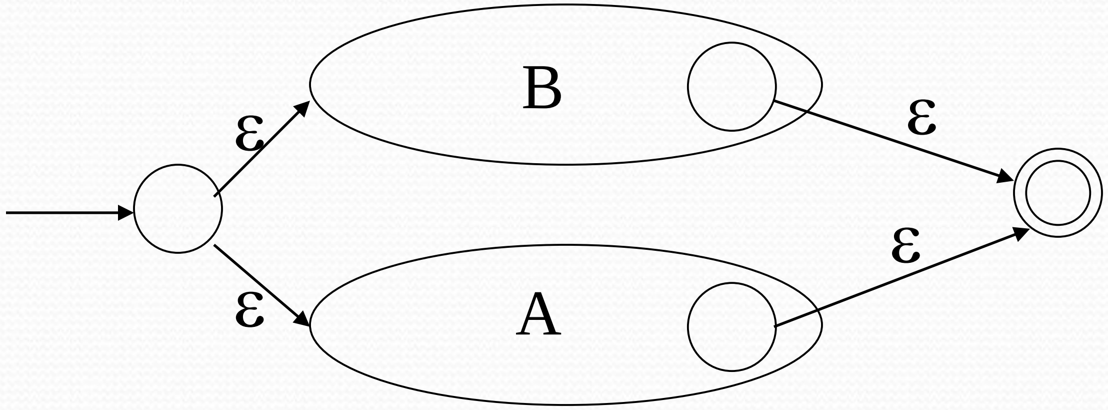


Regular Expressions to NFA (2)

- For AB

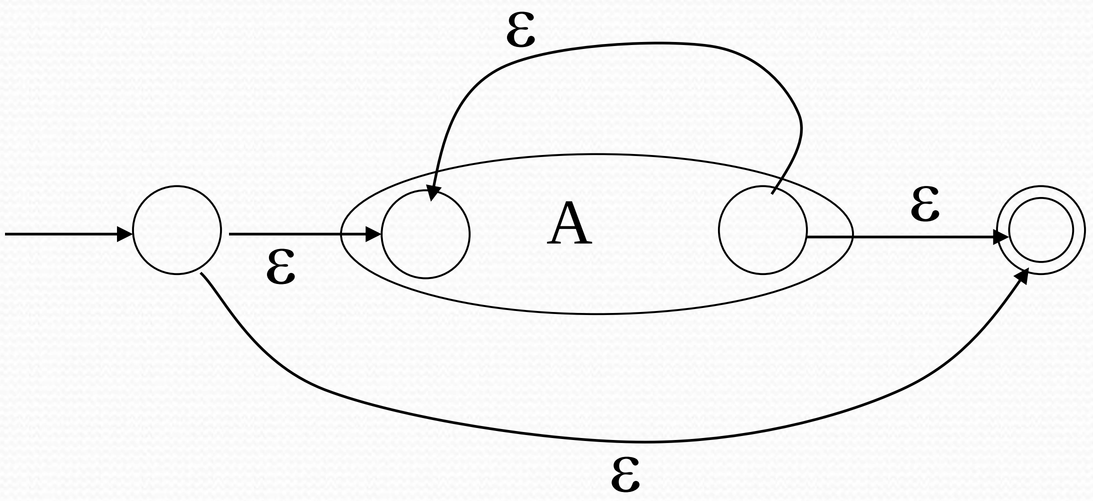


- For A | B



Regular Expressions to NFA (3)

- For A*



Examples

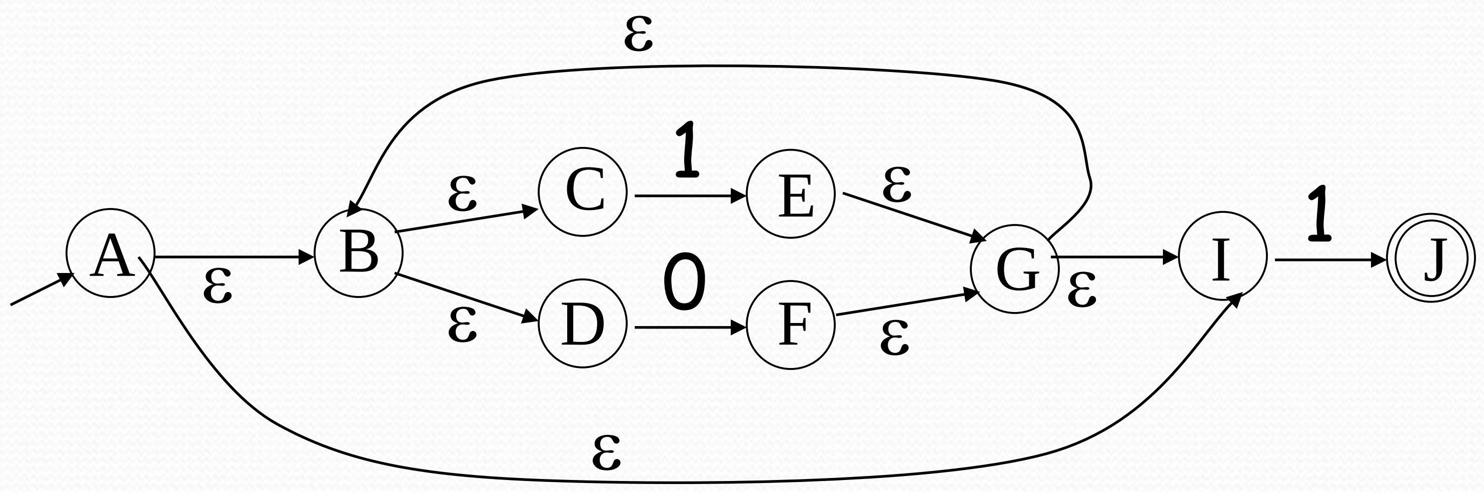
- Consider the regular expression
- a|(b)*
- aa*|bb*
- (ab)*a|b
- (0|1)*1

Example of RegExp -> NFA conversion

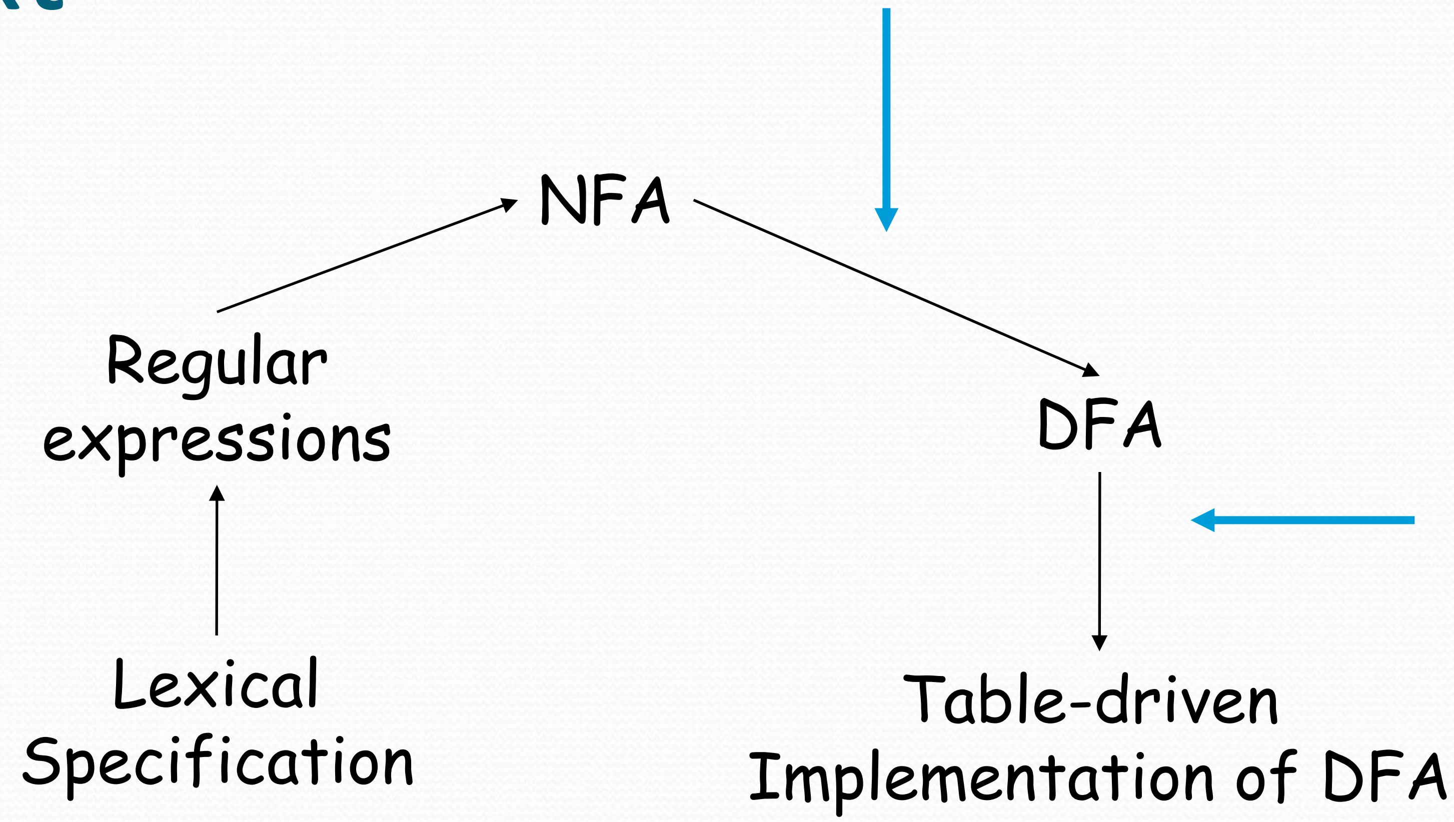
- Consider the regular expression

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

- The NFA is



Next


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

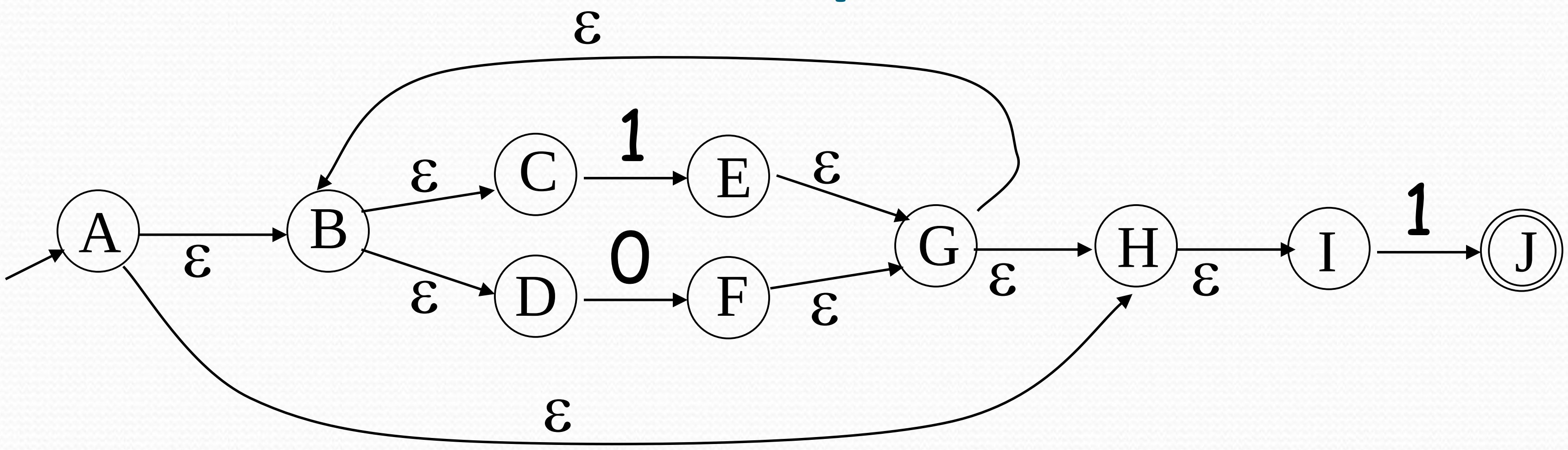
NFA to DFA. The Trick

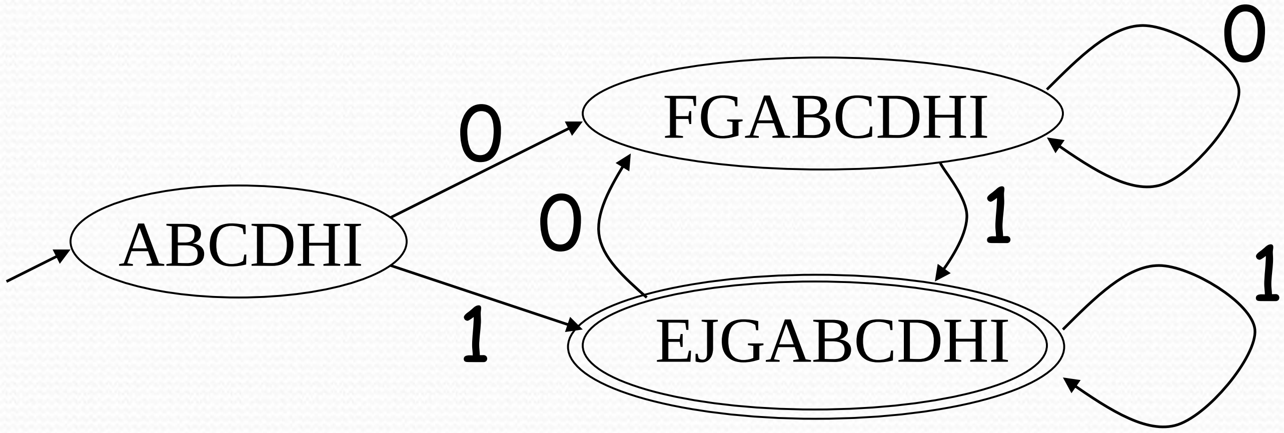
- Simulate the NFA
- Each state of resulting DFA

  = 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

NFA -> DFA Example


A
B
C
D
E
F
G
H
I
J
ε
1
0


ABCDHI
FGABCDHI
EJGABCDHI
0
1