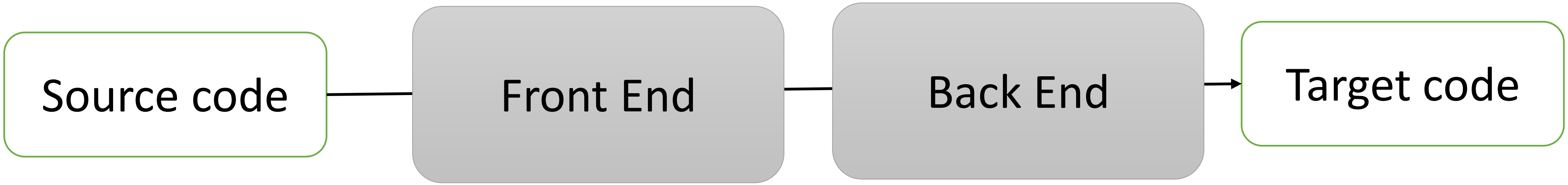
Chapter 2 – Lexical Analysis

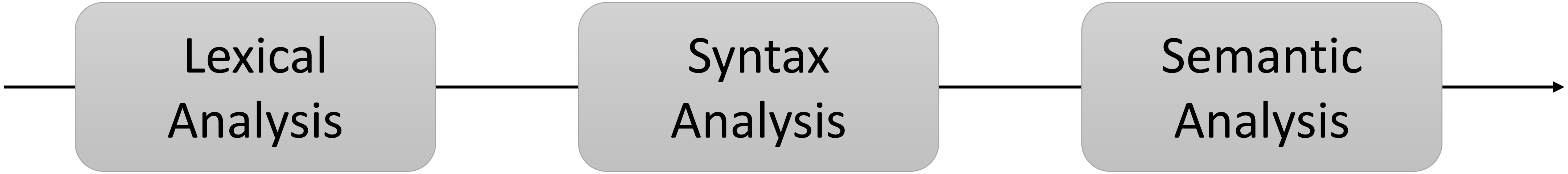
Compiler

- Compiler translates from one language to another



- Front End: Analysis
  - Takes input source code
  - Returns Abstract Syntax Tree and symbol table
- Back End: Synthesis
  - Takes AST and symbol table
  - Returns machine-executable binary code, or virtual machine code

Front End



- Lexical Analysis: breaks input into individual words – “tokens”
- Syntax Analysis: parses the phrase structure of program
- Semantic Analysis: calculates meaning of program

The role of the Lexical Analysis

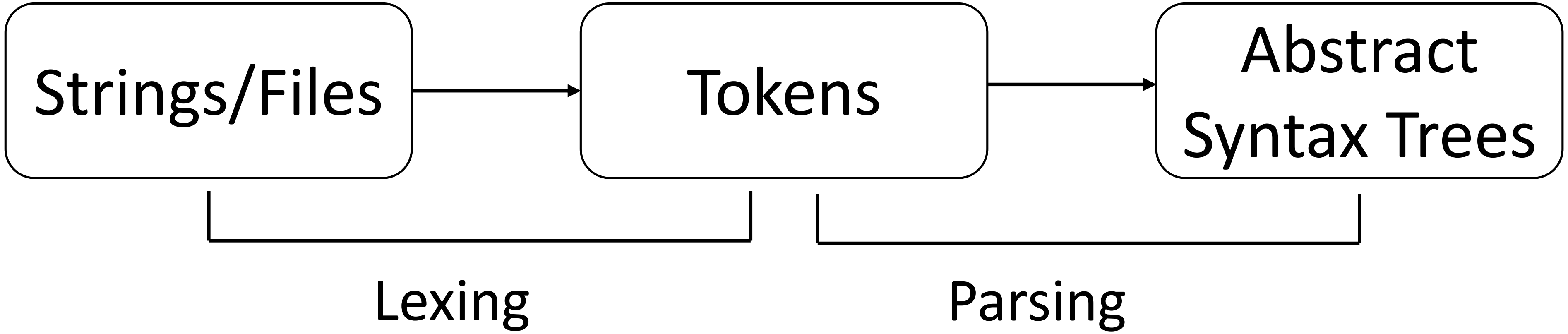
-> read the input characters of the source program

-> group them into lexemes

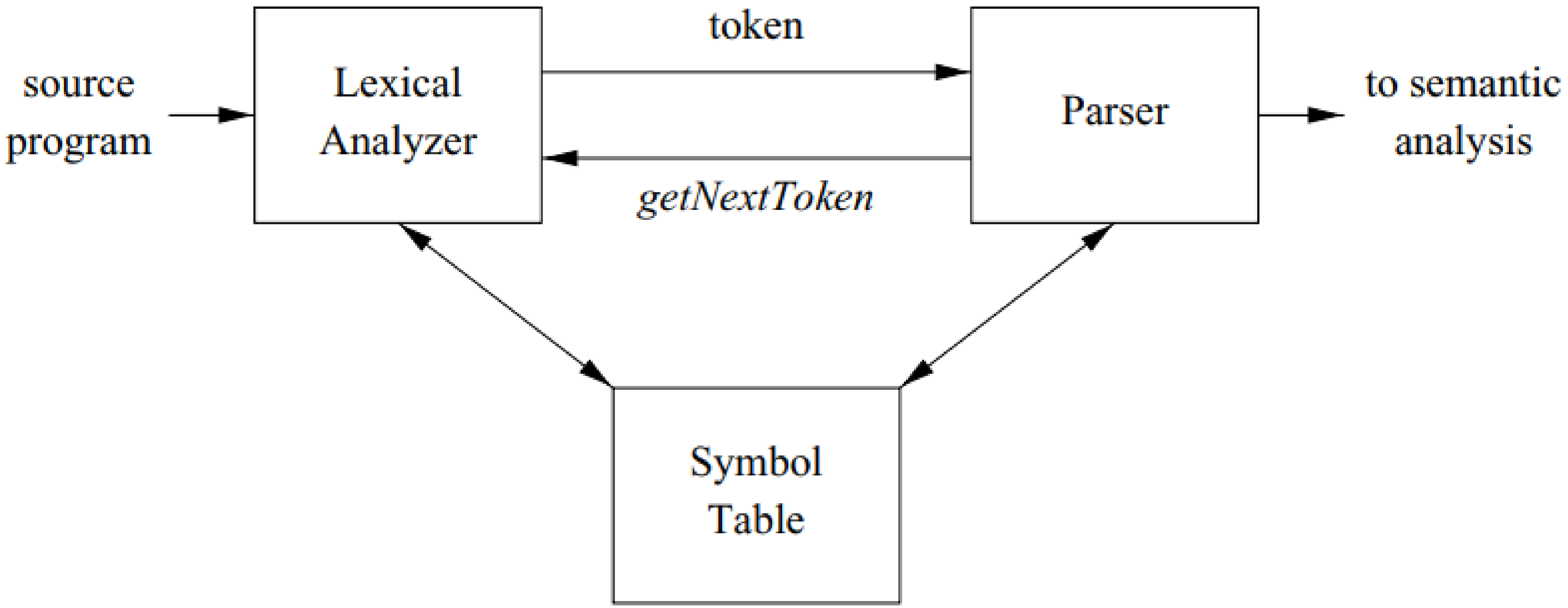
-> produce as output a sequence of tokens for each lexeme in the source program

Lexing & Parsing

- From strings to data structures



Interactions between the lexical analyzer and the parser



Tokens, Patterns and Lexemes

- A pattern is a description of the form that the lexemes of a token may take (the set of rule that define a TOKEN).

- A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identfied by the lexical analyzer as an instance of that token

- A *token* is a pair consisting of a token name and an optional attribute value.
  - Common token names are
    - identifiers: names the programmer chooses
    - keywords: names already in the programming language
    - separators (also known as punctuators): punctuation characters and paired-delimiters
    - operators: symbols that operate on arguments and produce results
    - literals: numeric, logical, textual, reference literals
    - ………..

Tokens, Patterns and Lexemes

- Consider this expression in the programming language C:

  sum=3+2;

- Tokenized and represented by the following table:

Lexeme	Token Name
sum	Identifier
=	Operator
3	Literal
+	Operator
2	Literal
;	Seperator

Tokens, Patterns and Lexemes

if (y <= t) y = y - 3;

Lexeme	Token Name
if	Keyword
(	Open parenthesis
y	Identifier
<=	Comparison operator
t	Identifier
)	Close parenthesis
y	Identifier
=	Assignment operator
y	Identifier
-	Arithmatic operator
3	Integer
;	semicolon

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.

- For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program.

Tokens, Patterns and Lexemes

cout << 3+2+3;

Lexeme	The following tokens are returned by scanner to parser in specified order
cout	<identifier, ‘cout’>
<<	<operator, ‘<<‘>
3	<literal, ‘3’>
+	<operator, ‘+’>
2	<literal, ‘2’>
+	<operator, ‘+’>
3	<literal, ‘3’>
;	<punctuator, ‘;’>

Tokens

```
if (num1 == num2)
    result = 1;
else
    result = 0;
```

```
\tif (num1 == num2)\n\t\tresult = 1;\n\telse\n\t\tresult = 0;
```

Tokens

- Token class
  - In English: noun, verb, adjective, …..
  - In a programming language: identifier, keyword, (,), number, …

Tokens

- Token classes correspond to sets of strings.

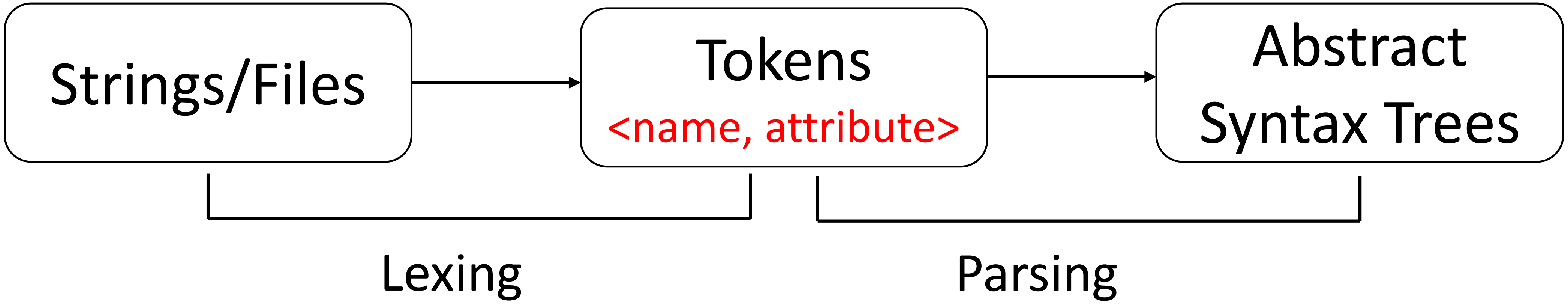
- Identifier:
  - Identifiers are strings of letters, digits, and underscores, starting with a letter or an underscore

    num1, result, name20, _result, …..
- Integer:
  - A non-empty string of digits

  10, 89, 001, 00, …….
- Keyword:
  - A fix set of reserved words

  if, else, for, while, ….
- Whitespace:
  - A non-empty sequence of blanks, newlines, and tabs

Lexical Analysis



Lexical Analysis

`result=50`

<id, ‘result’>
<op, ‘=‘>
<int, ’50’>

Abstract Syntax Trees

Lexing

Parsing

Lexical Analysis

```
\tif (num1 == num2)\n\t\tresult =  1;\n\telse\n\t\tresult = 0;
```

```
=> Go through and identify the tokens of the substrings.
```

`Whitespace:` A non-empty sequence of blanks, newlines, and tabs

`Keywords:` A fix set of reserved words

`Identifiers:` Identifiers are strings of letters, digits, and underscores, starting with a letter or an underscore

`Numbers`

`Operators`

`OpenParenthesis`

`CloseParenthesis`

`Semicolon`

Lexical Analysis: Regular expression

- Lexical structure = token classes

- Token classes correspond to sets of strings.
  - Use regular expressions to specify which set of strings belongs to each token class

Lexical Analysis: Regular expressions

- Single character
  ‘a’ = {“a”}
- Epsilon
  ε = {“”}
- Union
  $A + B = \{a \mid a \in A\} \cup \{b \mid b \in B\}$
- Concatenation
  $AB = \{ab \mid a \in A \wedge b \in B\}$
- Iteration
  $A^* = \bigcup_{i \geq 0} A^i$, $A^i = A \ldots\ldots\ldots A \ (i \ times)$
  $A^0 = \varepsilon$

Lexical Analysis: Regular expressions

- The regular expression over Σ are the smallest set of expressions including

$$
\begin{array}{lcll}
R & = & \varepsilon & \\
 & | & \text{'a'} & \text{where } c \in \Sigma \\
 & | & A + B & \text{where } A, B \text{ are regular expressions over } \Sigma \\
 & | & AB & \text{where } A, B \text{ are regular expressions over } \Sigma \\
 & | & A^* & \text{where } A \text{ is a regular expression over } \Sigma
\end{array}
$$

Lexical Analysis: Regular expressions

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

$1^* = \cup_{i \geq 0} 1^i =$ $\varepsilon + 1 + 11 + 111 + 1111 + \ldots\ldots.$

$(1+0)1 =$ $\{ab \mid a \in 1{+}0 \wedge b \in 1\}$ $= 11 + 01$

$0^* + 1^* =$ $\{0^\wedge i \mid i \geq 0\} \cup \{1^\wedge i \mid i \geq 0\}$ $= \varepsilon + 0 + 00 + 000 + 0000 + \ldots\ldots\ldots$
$+ \varepsilon + 1 + 11 + 111 + 1111 + \ldots\ldots.$

$(0+1)^* = \cup_{i \geq 0} (0 + 1)^i$

$= \varepsilon + (0{+}1) + (0{+}1)\,(0{+}1) + (0{+}1) \ldots\ldots (0{+}1)$

= all strings of 0's and 1's
$= \Sigma^*$

Lexical Analysis

Meaning function L maps syntax to semantics

$$L(e) = M$$

Regular expression

Set of strings

L(regular_expression)

L(regular_expression) -> set of strings

'a' = {"a"}	=> L('a') = {"a"}
ε = {""}	=> L(ε) = {""}
A + B = A ∪ B	=> L(A + B) = L(A) ∪ L(B)
AB = {ab | a∈A ∧ b ∈B}	=> L(AB) = {ab | a∈L(A) ∧ b ∈L(B)}
$A^* = \bigcup_{i \ge 0} A^i$,	=> $L(A^*) = \bigcup_{i \ge 0} L(A^i)$

Regular Expression

- keyword: A fix set of reserved words ("if" or "else" or "for" or …..)

Regular expression for if: 'i''f'
Regular expression for else: 'e''l''s''e'
Regular expression for for: 'f''o''r'

Regular expression for keyword:
'i''f' + 'e''l''s''e' + 'f''o''r' + ……….

=> 'if' + 'else' + 'for' + ……….

Regular Expression

- Integer: a non-empty string of digits

  - regular expression for the set of strings corresponding to all the single digits

$$\text{digit} = \text{‘0’} + \text{‘1’} + \text{‘2’} + \text{‘3’} + \text{‘4’} + \text{‘5’} + \text{‘6’} + \text{‘7’} + \text{‘8’} + \text{‘9’}$$

$$\text{integer} = \quad \text{digit digit}^* \quad = \text{digit}^+$$

Identifier: strings of letters, digits, and underscores, starting with a letter or an underscore.

digit = '0' + '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9'

= [0-9]

letter_ = [a-zA-Z_]

identifier = letter_(letter_ + digit)*

Whitespace: a non-empty sequence of blanks, newlines, and tabs

$$\text{whitespace} = (\text{' '} + \text{'\textbackslash n'} + \text{'\textbackslash t'})^{+}$$

student@vku.udn.vn

=> Make regular expression for this email address:

$\text{letter}^{+}\text{'@'}\text{letter}^{+}\text{'.'}\text{letter}^{+}\text{'.'}\text{letter}^{+}$

Regular Expression

- At least one: AA^* $\equiv A^+$
- Union: $A \mid B$ $\equiv A + B$
- Option: $A + \varepsilon$ $\equiv A?$
- Range: ‘a’ + ‘b’ + …+ ‘z’ ≡ [a-z]
- Excluded range: complement of [a-z] ≡ [^a-z]

Number in Pascal: A floating point number can have some digits, an optional fraction and an optional exponent (3.15E+10, 8E-3, 15.6, …)

digit = ‘0’+’1’+’2’+’3’+’4’+’5’+’6’+’7’+’8’+’9’

digits = digit^{+}

opt_fraction = (‘.’digits) + ε = (‘.’digits)?

opt_exponent = (‘E’(‘+’ + ’-’ + ε)digits) + ε = (‘E’(‘+’ + ‘-’)?digits)?

num = digits opt_fraction opt_exponent

Regular Expression

- Regular expressions describe many useful languages

- Regular languages are a language specification
  - We still need an implementation

Regular Expressions => Lexical Spec

1. Write a regular expressions for the lexemes of each token class

- number = digit^{+}
- keyword = ‘if’ + ‘else’ + …
- identifier = letter_(letter_ + digit)*
- openPar = ‘(‘
- closePar = ‘)’
- ………..

2. Construct R, matching all lexemes for all tokens

R = keyword + identifier + number + …..

  = R1 + R2 + ….

- (This step is done automatically by tools like flex)

3. Let input be $x_1 \ldots x_n$

For $1 \leq i \leq n$ check $x_1 \ldots x_i \in L(R)$?

4. If success, then we know that

$x_1 \ldots x_i \in L(R_j)$ for some j

R = R1 + R2 + R3 + …..

5. Remove $x_1 \ldots x_n$ from input and go to (3)

How much input is used?

If $x_1 \ldots . x_i \in L(R)$

And $x_1 \ldots . x_j \in L(R)$

$i \neq j$

Rule: Pick longest possible string in L(R)
– Pick k if k > i
– The “maximal munch”

Which token is used?

$x_1 \ldots. x_i \in L(R_j)$

$x_1 \ldots. x_i \in L(R_k)$ => which token is used?

Keywords = ‘if’ + ‘else’ + ….

Identifiers = letter(letter + digit)*

if $\in L(\text{Keywords})$

if $\in L(\text{Identifiers})$

=> Choose the rule listed FIRST.

- What if no rule matches?

$x_1 \ldots. x_i \notin L(R)$

Error = all strings not in the language of our lexical specification

Make a regular expression for error strings and PUT IT LAST IN PRIORITY (lowest priority)

- Regular expressions are a concise notation for string patterns

- Use in lexical analysis requires small extensions
  - To resolve ambiguities
    - Matches as long as possible
    - Highest priority match
  - To handle errors
    - Make a regular expression for error strings and PUT IT LAST IN PRIORITY.

Make a regular expression for:

- Keyword is a reserved word whose meaning is already defined by the programming language. We cannot use keyword for any other purpose inside programming. Every programming language have some set of keywords.

  Examples: int, do, while, void, return, ………..

Make a regular expression for:

- Identifiers

Identifiers are the name given to different programming elements. Either name given to a variable or a function or any other programming element, all follow some basic naming conventions listed below:

1. Keywords must not be used as an identifier.
2. Identifier must begin with an alphabet a-z A-Z or an underscore_ symbol.
3. Identifier can contains alphabets a-z A-Z, digits 0-9 and underscore _ symbol.
4. Identifier must not contain any special character (e.g. !@$*.'[] etc.) except underscore _.

Make a regular expression for:

- Operator

Operators are the symbol given to any arithmetical or logical operations. Various programming languages provides various sets of operators some common operators are:

  - Arithmetic operator (+, -, *, / %)
  - Assignment operator (=)
  - Relational operator (>, <, >=, <=, ==, !=)
  - Logical operator (&&, ||, !)
  - Bitwise operator (&, |, ^, ~, <<, >>)
  - Increment/Decrement operator (++, --)
  - ~~Conditional/Ternary operator (? :)~~

Make a regular expression for:

- Literals

Literals are constant values that are used for performing various operations and calculations. There are basically three types of literals:

1. Integer literal

   An integer literal represents integer or numeric values.

   Example: 1, 100, -12312 etc

2. Floating point literal

   Floating point literal represents fractional values.

   Example: 2.123, 1.02, -2.33, 13e54, -23.3 etc

3. Character literal

   Character literal represent character values. Single character are enclosed in a single quote(' ') while sequence of character are enclosed in double quotes(" ")

   Example: 'a', 'n', "Hello", "Hello123" etc.

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automata consists of
  - An input alphabet Σ
  - A finite set of states S
  - A start state q0
  - A set of accepting states $F \subseteq S$
  - A set of transitions δ

$$\text{state} \xrightarrow{\text{input}} \text{state}$$

Finite Automata

- Transition

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

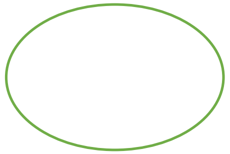
- Is read

  In state s_1 on input a go to state s_2

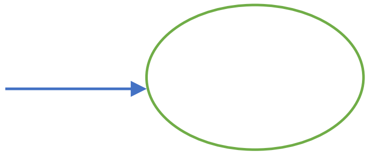
- If end of input and in accepting state => accept
- Otherwise => reject
  - Terminates in state $s \notin F$
  - Get stuck

Finite Automata

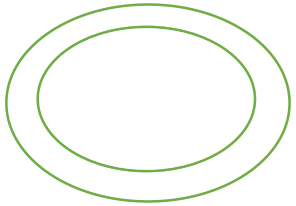
- A state



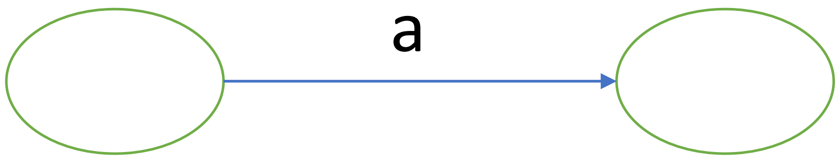
- The start state



- An accepting state

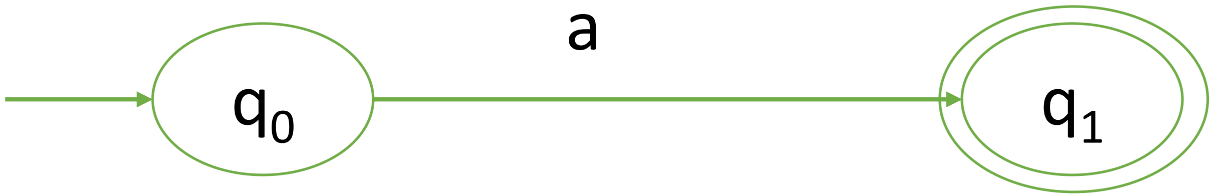


- A transition



Finite Automata

- A finite automata that accepts only “a”

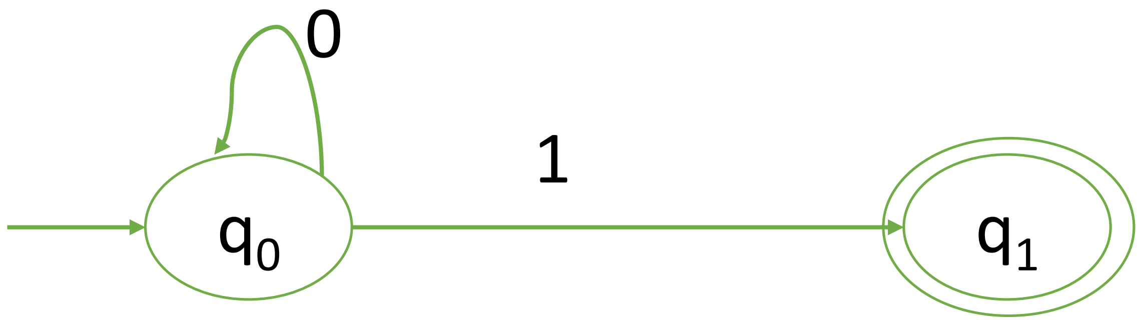


- What happen if input strings are:
  - “a”
  - “b”
  - “ab”

- Language of a finite automata is set of accepted strings.

Finite Automata

- A finite automata accepting any number of 0's followed by a single 1.



STATE	INPUT
q_0	0 0 1
q_0	0 0 1
q_0	0 0 1
q_1	0 0 1

Accept

STATE	INPUT
q_0	0 1 1
q_0	0 1 1
q_1	0 1 1

Reject

Regular Expressions to non-deterministic finite automata (NFA)

Lexical Specification

Regular Expressions

Non-deterministic Finite Automata (NFA)

Deterministic Finite Automata (DFA)

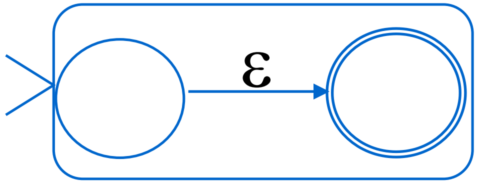
Table-driven Implementation of DFA

Regular Expressions to NFA

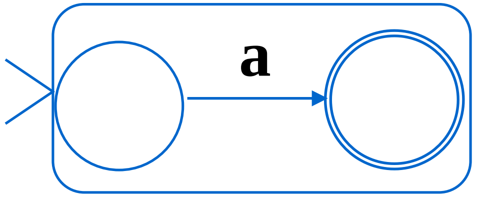
- For each kind of regular expression, define an equivalent NFA that accepts exactly the same language as the language of a regular expression.

  ⇒NFA for regular expression M

- For ε

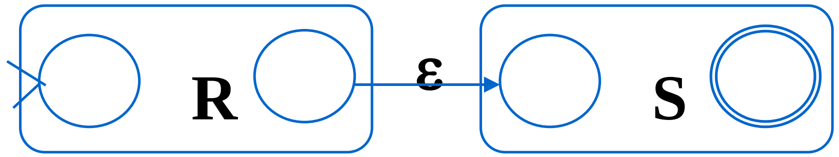


- For input a

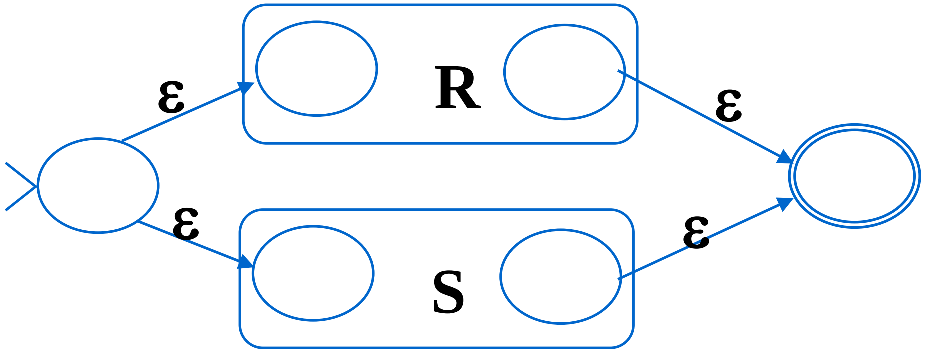


Regular Expressions to NFA

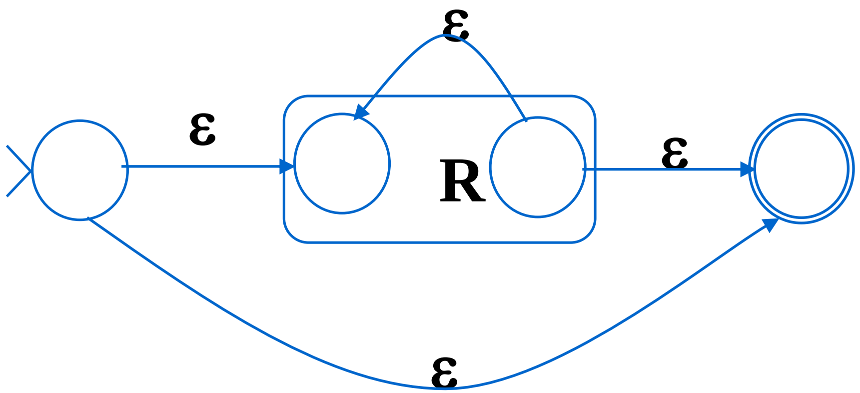
- Concatenation
  - For RS



- Union
  - For R + S

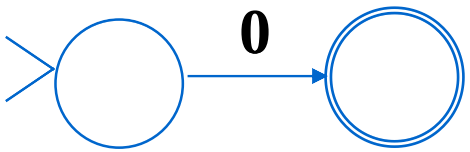


- Iteration
  - For R*

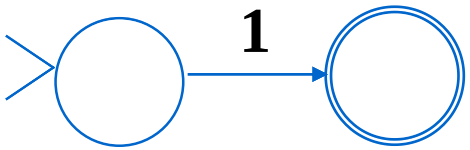


Regular Expressions to NFA

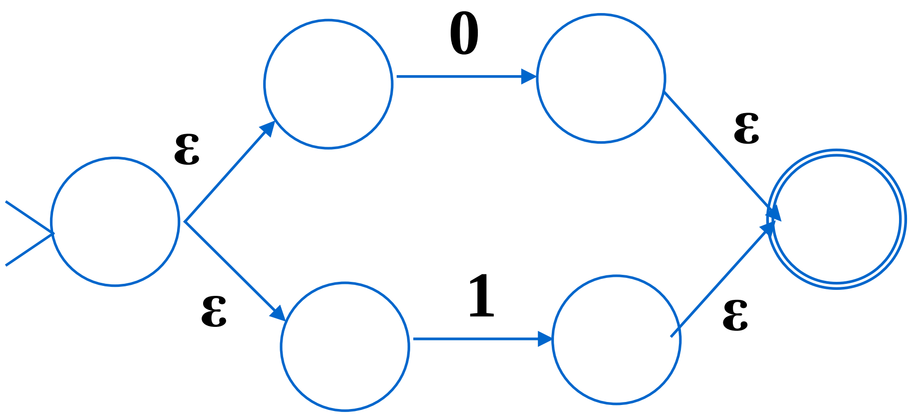
- Consider the regular expression (0+1)(01)*
  - For 0



  - For 1

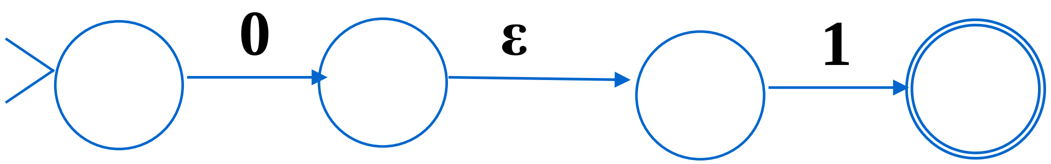


  - For 0 + 1

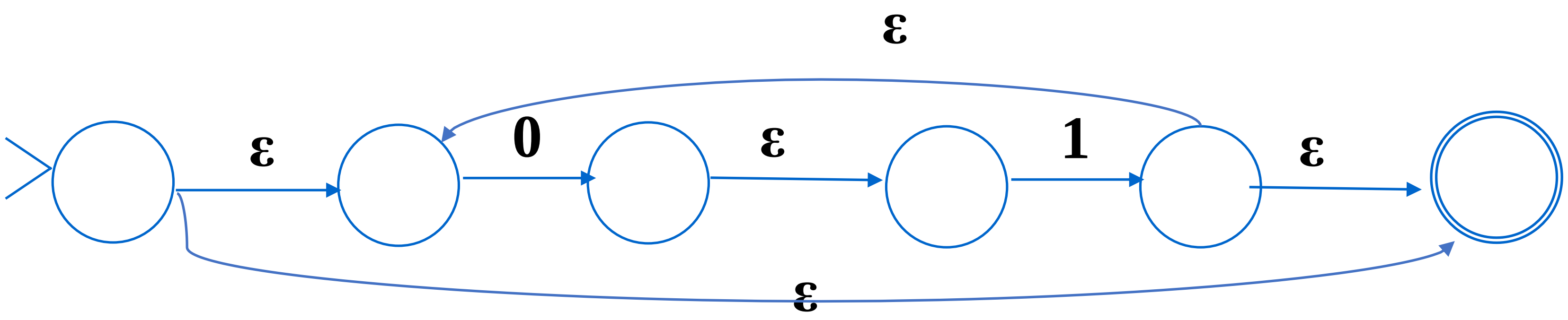


Regular Expressions to NFA

- Consider the regular expression (0+1)(01)*
  - For 01

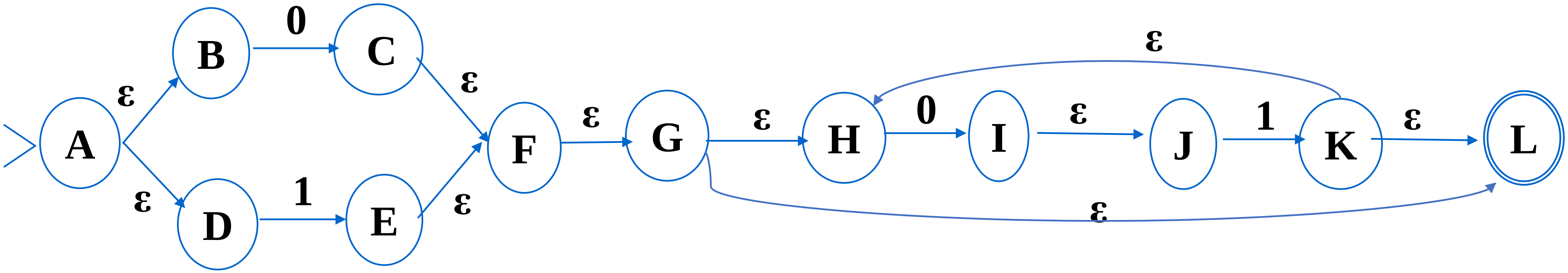


  - For (01)*

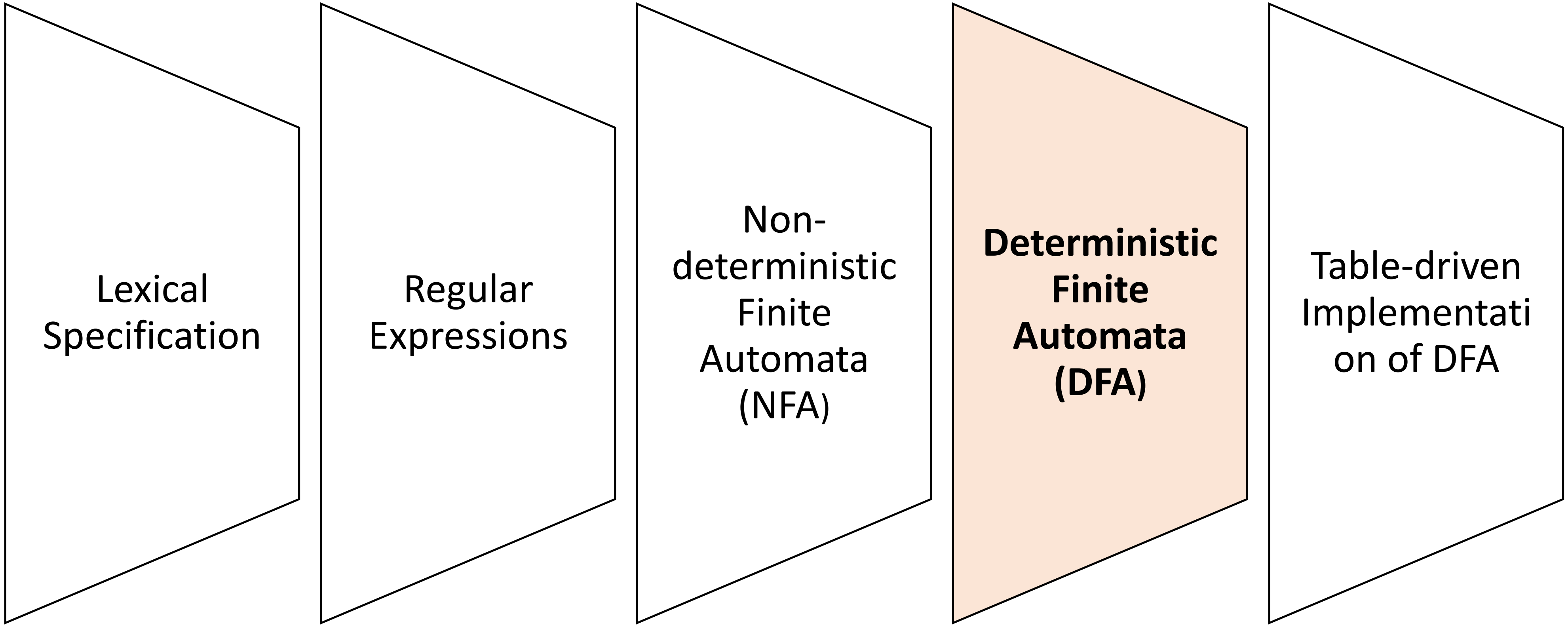


Regular Expressions to NFA

- Consider the regular expression (0+1)(01)*



Regular Expressions to non-deterministic finite automata (NFA)



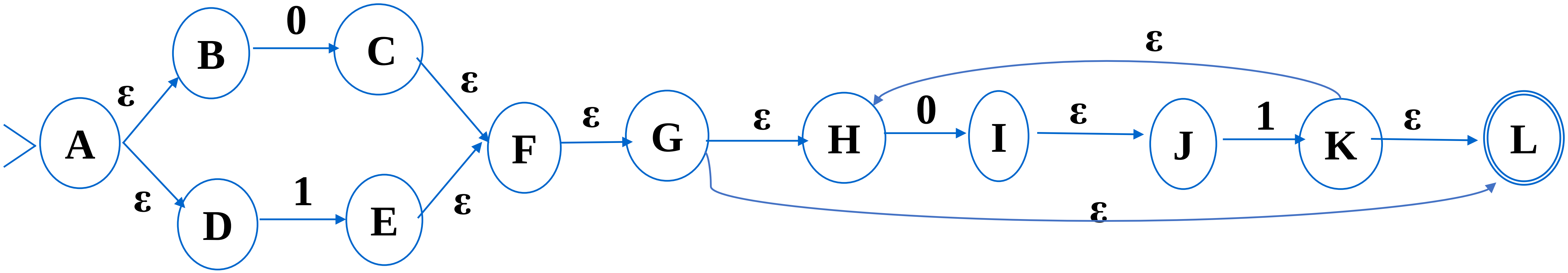
NFA to DFA

- Simulate the NFA
- Each state of DFA
  = a non-empty subset of states of the NFA
- Start state of DFA
  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \xrightarrow{a} S'$ to DFA if
  – S' is the set of NFA states reachable from any
  state in S after seeing the input a, considering ε-moves as well
- Final state of DFA

  = the set includes the final state of the NFA

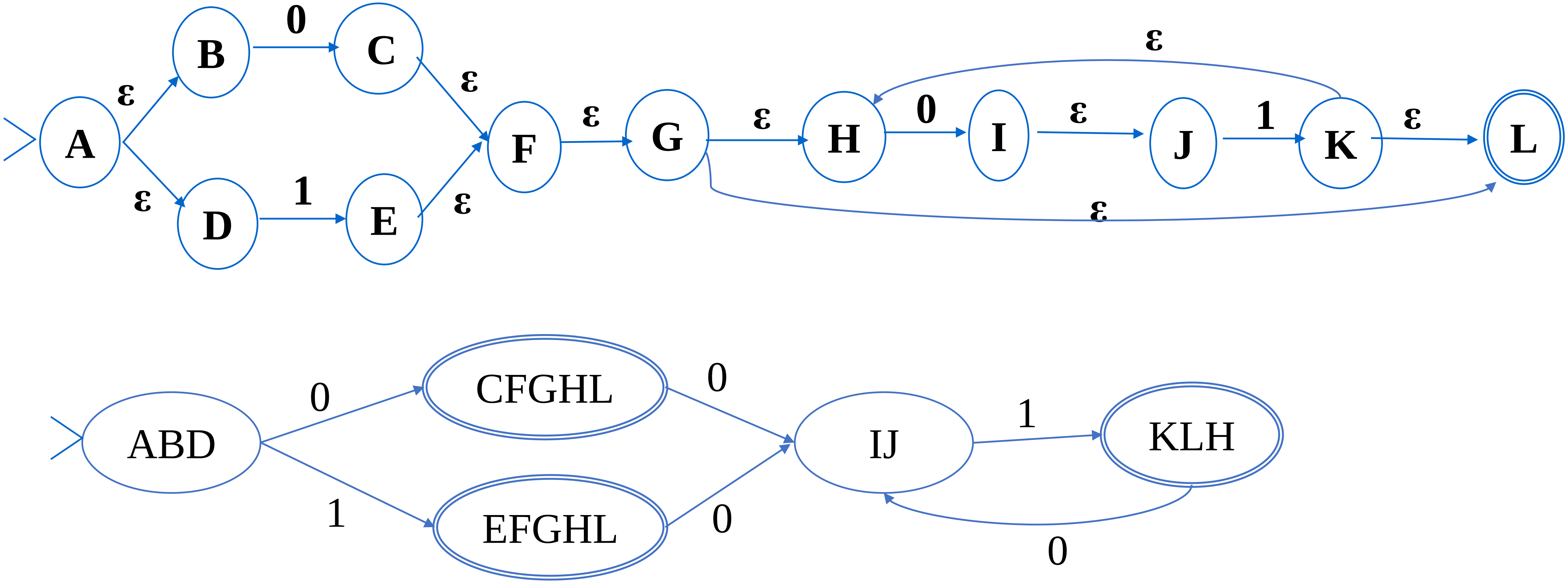
NFA to DFA

- NFA for (0+1)(01)*



NFA to DFA

- NFA for (0+1)(01)*



Regular Expressions to non-deterministic finite automata (NFA)

Lexical Specification

Regular Expressions

Non-deterministic Finite Automata (NFA)

Deterministic Finite Automata (DFA)

Table-driven Implementation of DFA

Implementation of DFA

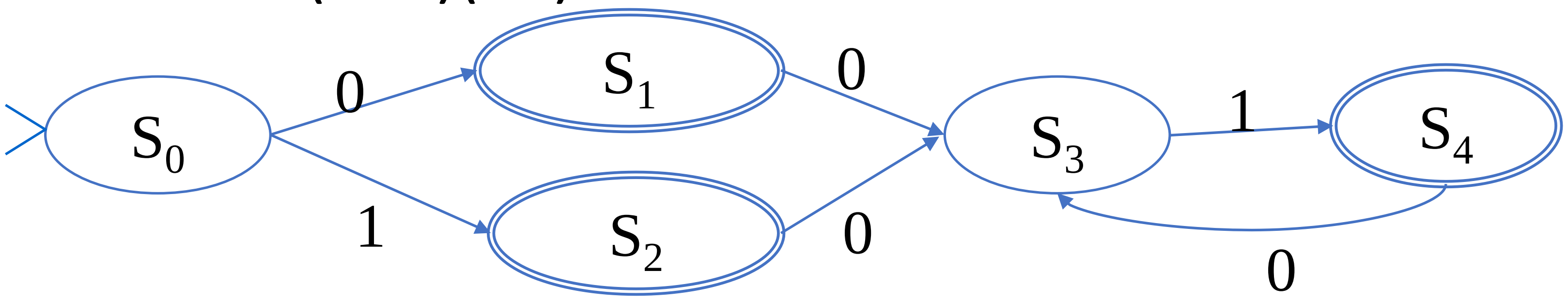
- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbol"
  - For every transition $S_i \xrightarrow{a} S_k$ define T[i,a] = k

	Input symbols	
states		

	a	b
i	k	
j		
k		
l		

Implementation of DFA

- DFA for (0+1)(01)*



	0	1
S_0	S_1	S_2
S_1	S_3	
S_2	S_3	
S_3		S_4
S_4	S_3	

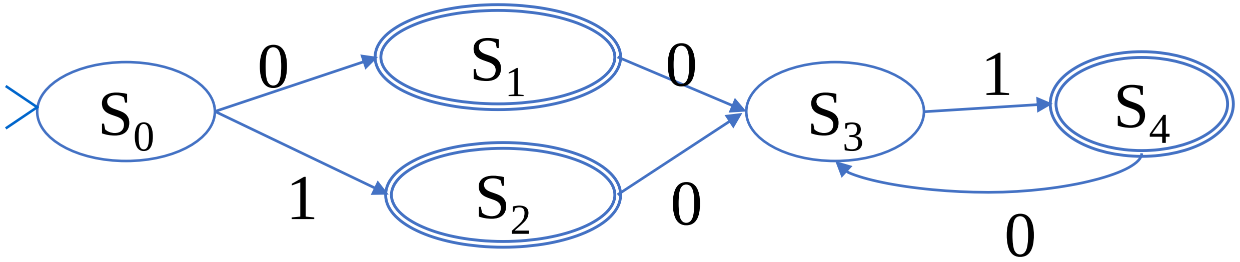
Implementation of DFA

```
i = 0;
state = 0;
while (input[i]){
   state = T[state, input[i++]];
}
```

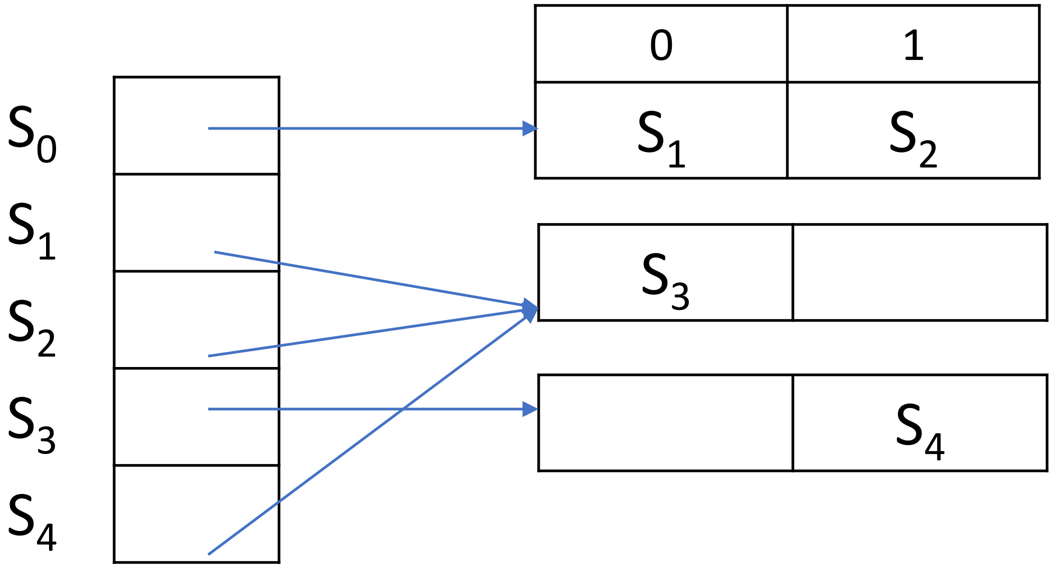
	0	1
S_0	S_1	S_2
S_1	S_3	
S_2	S_3	
S_3		S_4
S_4	S_3	

Implementation of DFA

- DFA for (0+1)(01)*

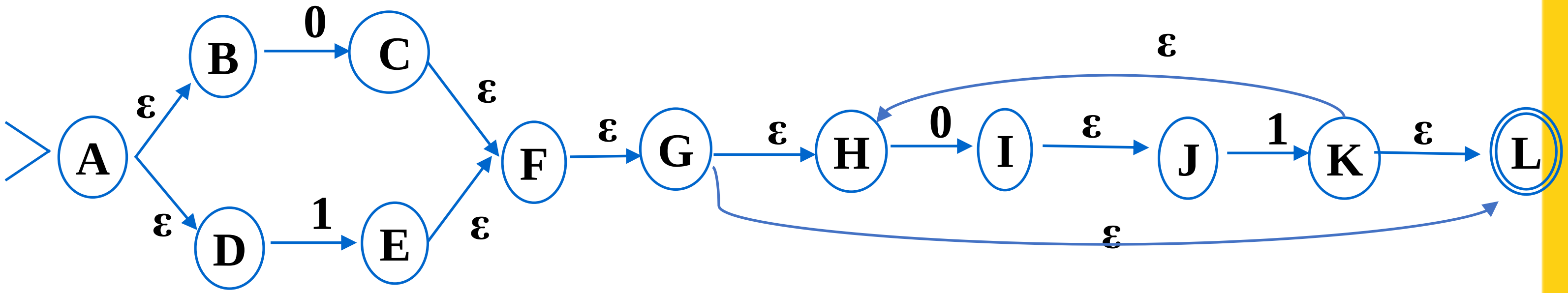


	0	1
S_0	S_1	S_2
S_1	S_3	
S_2	S_3	
S_3		S_4
S_4	S_3	



Implementation of NFA

	0	1	ε
A			{B, D}
B	{C}		
C			{F}
D		{E}	
E			{F}
F			{G}
G			{H, L}
H	{I}		
I			{J}
J		{K}	
K			{L, H}



Summarize

- Conversion of NFA to DFA is the key
- DFAs are faster and less compact so the tables can be very large
- NFAs are slower to implement but more concise.
- In practice, tools provide tradeoffs between speed and space.
- Tools give generally a series of options in the form of configuration files or command lines which allow you to choose whether you want to be closer to a full DFA or to a pure NFA.

Assignment 1 (Lexical Analyzer)