

LEXICAL ANALYSIS

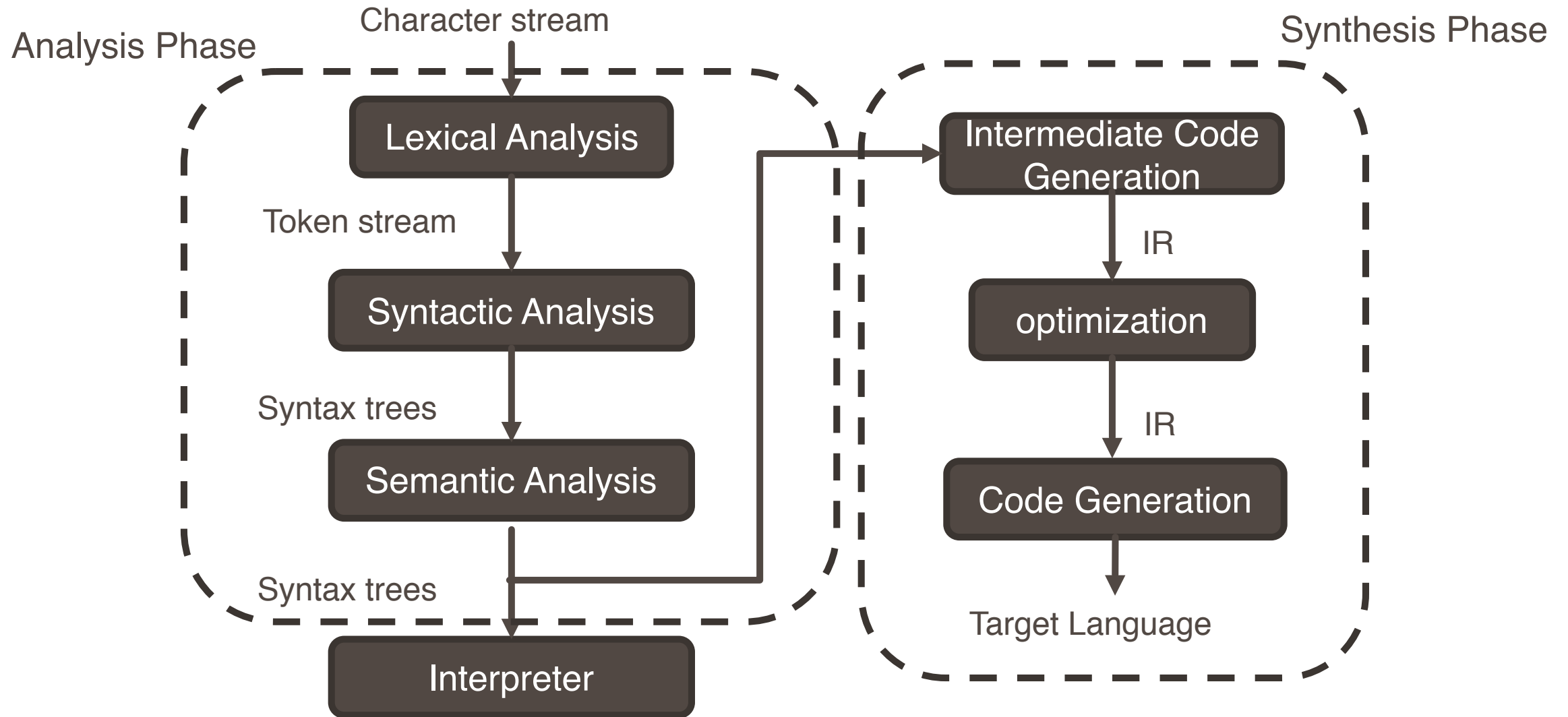
Baishakhi Ray

Fall 2019

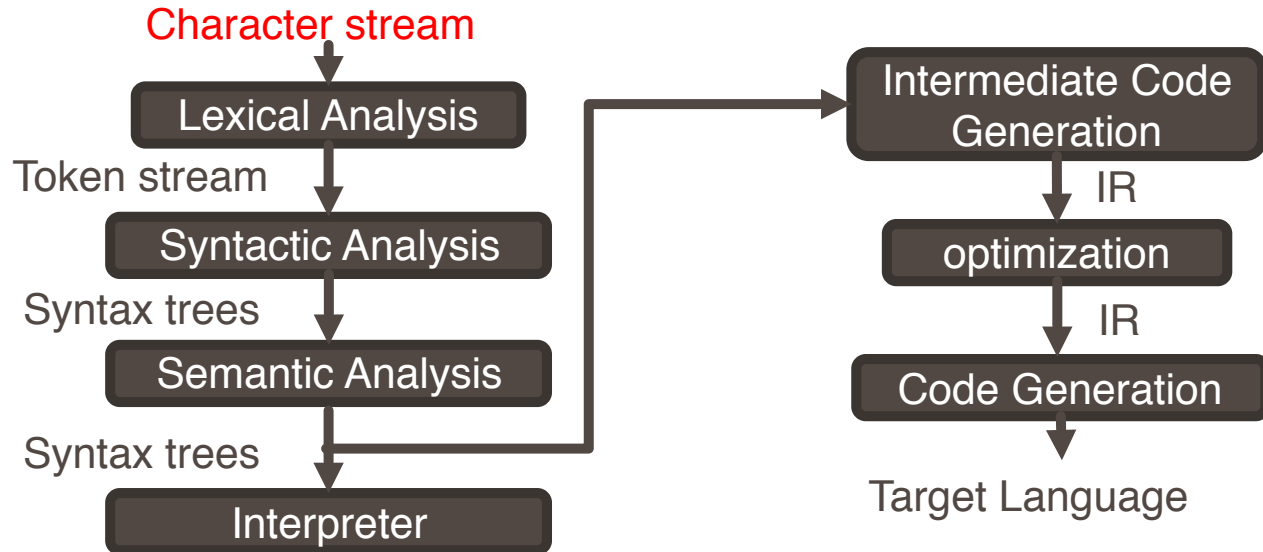
These slides are motivated from Prof. Alex Aiken: Compilers (Stanford)



Structure of a Typical Compiler



Input to Compiler

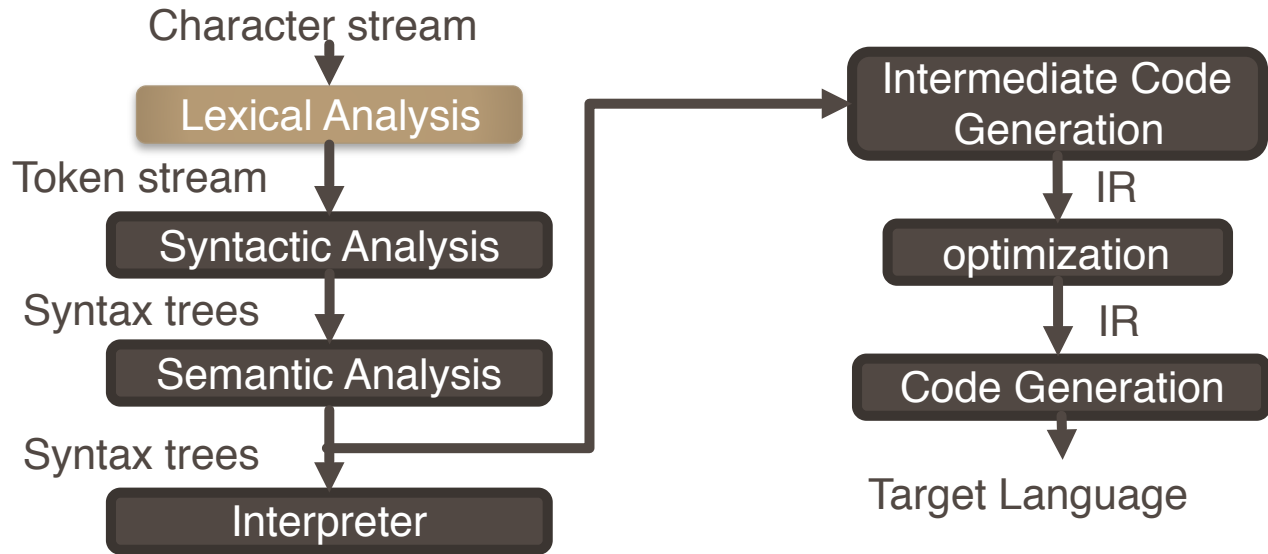


```
if(i == j)
z = 0;
else
z = 1;
```

```
if ( i == j )\n\t z = 0 ; \n else\n\t z = 1 ;
```

Lexical Analysis

Break character stream into tokens ("words")



```
if(i == j)
  z = 0;
else
  z = 1;
```

```
if ( i = = j ) \n \t z = 0 ; \n else \n \t z = 1 ;
```

```
'if' '(' 'i' '==' 'j' ')' '\n' '\t' 'z' '=' '0' ';' '\n' 'else' '\n' '\t' 'z' '=' '1' ';' ;
```

```
keyword<if> LPAR identifier<i> op<==> Identifier<j> RPAR
whitespaces identifier<z> op<=> number<0> <;> whitespaces
keyword<else> identifier<z> op<=> number<1> ';' ;
```

1. Identify the substrings

2. Identify the token classes

Token Class

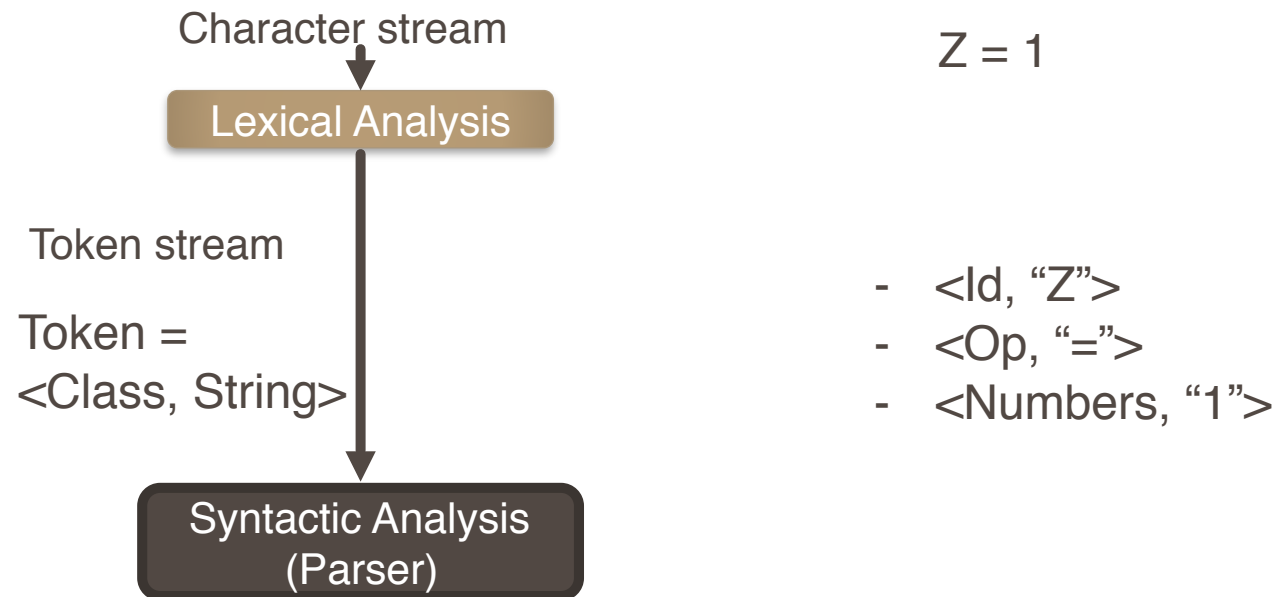
- In English?
 - Noun, verb, adjectives, ...
- In Programming Language
 - keywords, identifiers, LPAR, RPAR, const, etc.

Token Class

- Each class corresponds to a set of strings
- Identifier
 - Strings are letters or digits, starting with a letter
 - Eg:
- Numbers:
 - A non-empty strings of digits
 - Eg:
- Keywords
 - A fixed set of reserved words
 - Eg:
- Whitespace
 - A non-empty sequence of blanks, newlines, and tabs

Lexical Analysis (Example)

- Classify program substrings according to roles (token class)
- Communicate tokens to parser



"Z", "=", "1" are called lexemes (an instance of the corr. token class)

Lexical Analysis: HTML Examples

```
Here is a photo of <b> my house </b>
<p><br/>
see <a href="morePix.html">More Picture</a> if you liked that
one.</p>
```

```
<text, "Here is a photo of">
<nodestart, b>
<text, "my house">
<nodeend, b>
<nodestart, p>
<selfendnode, img>
<selfendnode, br>
<text, "see">
<nodestart, a>
<text, "More Picture">
<nodeend, a>
<text, "if you liked that one.">
<nodeend, p>
```


Exercise

```
x = p;  
while ( x < 100 ) { x++ ; }
```

Exercise

```
if(i == j)
  z = 0;
else
  z = 1;
```

==/=?

Keyword/Identifier?

Lookahead

- Lexical analysis tries to partition the input string into the logical units of the language. This is implemented by reading left to right. “scanning”, recognizing one token at a time.
- “Lookahead” is required to decide where one token ends and the next token begins.

```
if(i == j)
  z = 0;
else
  z = 1;
```

==/=?

Keyword/Identifier?

Lookahead: Examples

- Usually, given the pattern describing the lexemes of a token, it is relatively simple to recognize matching lexemes when they occur on the input.
- However, in some languages, it is not immediately apparent when we have seen an instance of a lexeme corresponding to a token.

FORTRAN RULE: White Space is insignificant: VA R1 == VAR1

DO 5 I = 1,25

DO 5 I = 1.25

- Lexical analysis may require to “look ahead” to resolve ambiguity.
 - Look ahead complicates the design of lexical analysis
 - Minimize the amount of look ahead

Lexical Analysis: Examples

- C++ template Syntax:
 - `Foo<Bar>`
- C++ stream Syntax:
 - `cin >> var`
- Ambiguity
 - `Foo<Bar<Barq>>`
 - `cin >> var`

Summary So Far

- The goal of Lexical Analysis
 - Partition the input string to lexeme
 - Identify the token class of each lexeme

- Left-to-right scan => look ahead may require
 - In reality, lookahead is always needed
 - Our goal is to minimize the amount of lookahead



REGULAR LANGUAGES

-
- Lexical structure of a programming language is a set of token classes.
 - Each token class consists of some set of strings.
 - How to map which set of strings belongs to which token class?
 - Use regular languages
 - Use Regular Expressions to define Regular Languages.

Regular Expressions

- Single character

- $'c' = \{“c”\}$

- Epsilon

- $\mathcal{E} = \{“”\}$

- 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 (Kleene closure)

- $A^* = \bigcup_{i \geq 0} A^i = A \dots A \text{ (i times)}$

- $A^0 = \mathcal{E}$ (empty string)

Regular Expressions

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

$R = \epsilon$

| 'c', 'c' $\in \Sigma$

| $R + R$

| RR

| R^*

Regular Expression Example

- $\Sigma = \{p,q\}$
 - q^*
 - $(p+q)q$
 - p^*+q^*
 - $(p+q)^*$

- There can be many ways to write an expression

Exercise

Choose the regular languages that are equivalent to the given regular language: $(p + q)^*q(p + q)^*$

A. $(pq + qq)^*(p + q)^*$

B. $(p + q)^*(qp + qq + q)(p + q)^*$

C. $(q + p)^*q(q + p)^*$

D. $(p + q)^*(p + q)(p + q)^*$

Formal Languages

- Def: Let Σ be a set of character (alphabet). A language over Σ is a set of strings of characters drawn from Σ .
 - Regular languages is a formal language
- Alphabet = English character, Language = English Language
 - Is it formal language?
- Alphabet = ASCII, Language = C Language

Formal Language

$$'c' = \{ "c" \}$$

$$\epsilon = \{ "" \}$$

$$A + B = \{ a \mid a \in A \} \cup \{ b \mid b \in B \}$$

$$AB = \{ ab \mid a \in A \wedge b \in B \}$$

$$A^* = \bigcup_{i \geq 0} A^i$$

expression $i \geq 0$ Set

Formal Language

$$L('c') = \{“c”\}$$

$$L(\epsilon) = \{“”\}$$

$$L(A + B) = \{a \mid a \in L(A)\} \cup \{b \mid b \in L(B)\}$$

$$L(AB) = \{ab \mid a \in L(A) \wedge b \in L(B)\}$$

$$L(A^*) = \bigcup_{i \geq 0} L(A^i)$$

expression Set

L: Expressions \rightarrow Set of strings

- Meaning function L maps syntax to semantics
- Mapping is many to one
- **Never one to many**

Lexical Specifications

- Keywords: “if” or “else” or “then” or “for”
 - Regular expression = ‘i’ ‘f’ + ‘e’ ‘l’ ‘s’ ‘e’
= ‘if’ + ‘else’ + ‘then’
- Numbers: a non-empty string of digits
 - digit = ‘1’+‘0’+‘2’+‘3’+‘4’+‘5’+‘6’+‘7’+‘8’+‘9’
 - digit*
 - How to enforce non-empty string?
 - digit digit* = digit+

Lexical Specifications

- Identifier: strings of letters or digits, starting with a letter
 - letter = 'a' + 'b' + 'c' + + 'z' + 'A' + 'B' + + 'Z'
= [a-zA-Z]
 - letter (letter + digit)*
- Whitespace: a non-empty sequence of blanks, newline, and tabs
 - (' ' + '\n' + '\t')+

PASCAL Lexical Specification

- $\text{digit} = '0' + '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9'$
- $\text{digits} = \text{digit}^+$
- $\text{opt_fraction} = ('.' \text{ digits}) + \varepsilon = ('.' \text{ digits})?$
- $\text{opt_exponent} = ('E' ('+' + '-' + \varepsilon) \text{ digits}) + \varepsilon$
 $= ('E' ('+' + '-')? \text{ digits})?$
- $\text{num} = \text{digits} \text{ opt_fraction} \text{ opt_exponent}$

Common Regular Expression

- At least one $A^+ \equiv AA^*$
- Union: $A \mid B \equiv A + B$
- Option: $A? \equiv A + \varepsilon$
- Range: $'a' + \dots + 'z' = [a-z]$
- Excluded range: complement of $[a-z] \equiv [^a-z]$

Summary of Regular Languages

- Regular Expressions specify regular languages
- Five constructs
 - Two base expression
 - Empty and 1-character string
 - Three compound expressions
 - Union, Concatenation, Iteration

Lexical Specification of a language

1. Write a regex for the lexemes of each token class
 - Number = digit⁺
 - Keywords = 'if' + 'else' + ..
 - Identifiers = letter (letter + digit)*
 - LPAR = '('

Lexical Specification of a language

2. Construct R , matching all lexemes for all tokens

$$R = \text{Number} + \text{Keywords} + \text{Identifiers} + \dots$$
$$= R_1 + R_2 + R_3 + \dots$$

3. Let input be $x_1 \dots x_n$.

For $1 \leq i \leq n$, check $x_1 \dots x_i \in L(R)$

4. If successful, then we know that

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

5. Remove $x_1 \dots x_i$ from input and go to step 3.

Lexical Specification of a language

- How much input is used?
 - $x_1 \dots x_i \in L(R)$
 - $x_1 \dots x_j \in L(R), i \neq j$
 - Which one do we want? (e.g., `==` or `=`)
 - **Maximal munch**: always choose the longer one
- Which token is used if more than one matches?
 - $x_1 \dots x_i \in L(R)$ where $R = R_1 + R_2 + \dots + R_n$
 - $x_1 \dots x_i \in L(R_m)$
 - $x_1 \dots x_i \in L(R_n), m \neq n$
 - Eg: Keywords = 'if', Identifier = letter (letter + digit)*, if matches both
 - Keyword has higher priority
 - Rule of Thumb: **Choose the one listed first**

Lexical Specification of a language

- What if no rule matches?
 - $x_1 \dots x_i \notin L(R)$... compiler typically tries to avoid this scenario
 - Error = [all strings not in the lexical spec]
 - Put it in last in priority

Summary so far

- Regular Expressions are concise notations for the string patterns
- Use in lexical analysis with some extensions
 - To resolve ambiguities
 - To handle errors
- Implementation?
 - We will study next

Finite Automata

- Regular Expression = specification
- Finite Automata = implementation

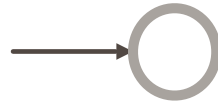
- A finite automaton consists of

- An input Alphabet: Σ

- A finite set of states: S



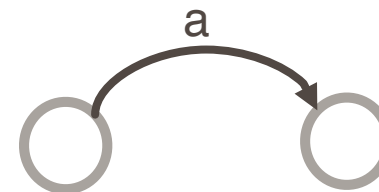
- A start state: n



- A set of accepting states: $F \subseteq S$



- A set of transitions state: state1 $\xrightarrow{\text{input}}$ state2



Transition

- $s1 \xrightarrow{a} s2$ (state $s1$ on input a goes to state $s2$)
- If end of the input and in final state, the input is accepted
- Otherwise reject

- Language of FA = set of strings accepted by that FA

Example Automata

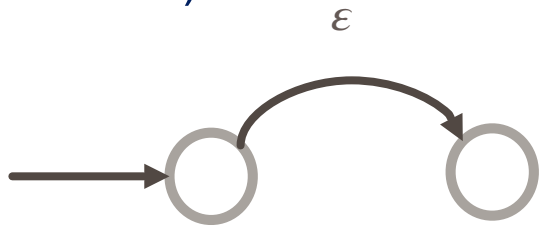
- a finite automaton that accepts only “1”

Example Automata

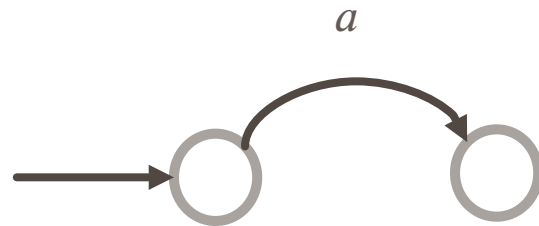
- A finite automaton that accepting any number of “1” followed by “0”

Regular Expression to NFA

- For ϵ (it's a choice)



- For input a

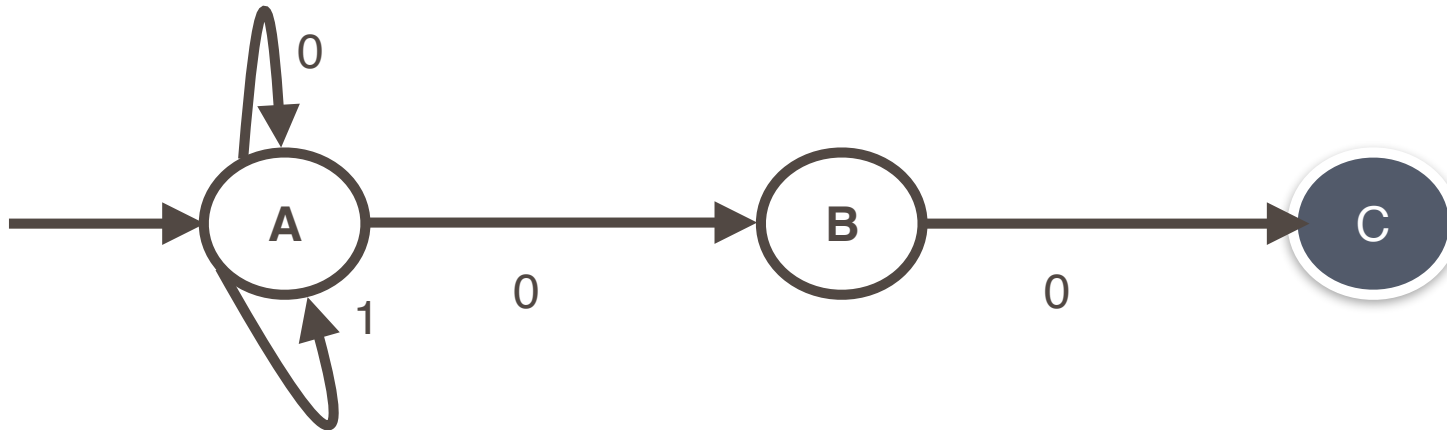


Finite Automata

- Deterministic Finite Automata (DFA)
 - One transition per input per state
 - No ϵ -moves
 - Takes only one path through the state graph
- Nondeterministic Finite Automata (NFA)
 - Can have multiple transitions for one input in a given state
 - Can have ϵ -moves
 - Can choose which path to take
 - An NFA accepts if some of these paths lead to accepting state at the end of input.

Finite Automata

- An NFA can get into multiple states



- Input: 1 0 0
- Output: {A}. {A,B} {A,B,C}

NFA vs. DFA

- NFAs and DFAs recognize the same set of regular languages
- DFAs are faster to execute
 - No choices to consider
- NFAs are, in general, small

Lexical Specification



Regular Expressions



NFA



DFA



Table driven implementation of automata

Lexical Specification



Regular Expressions



NFA



DFA



Table driven implementation of automata

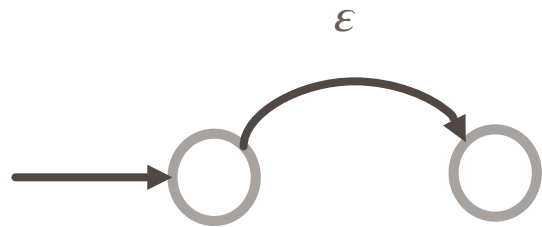
Finite Automata

- For each kind of regex, define an equivalent NFA
 - Notation: NFA for regex M

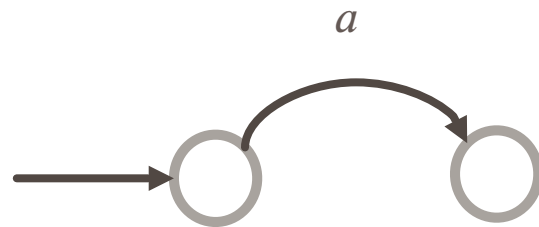


Regular Expression to NFA

- For ϵ



- For input a

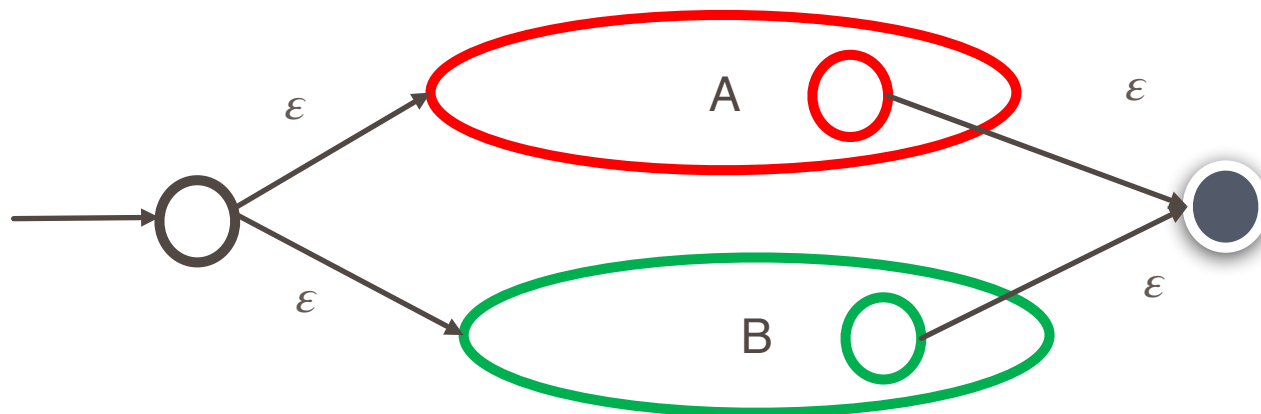


Regular Expression to NFA

- For AB

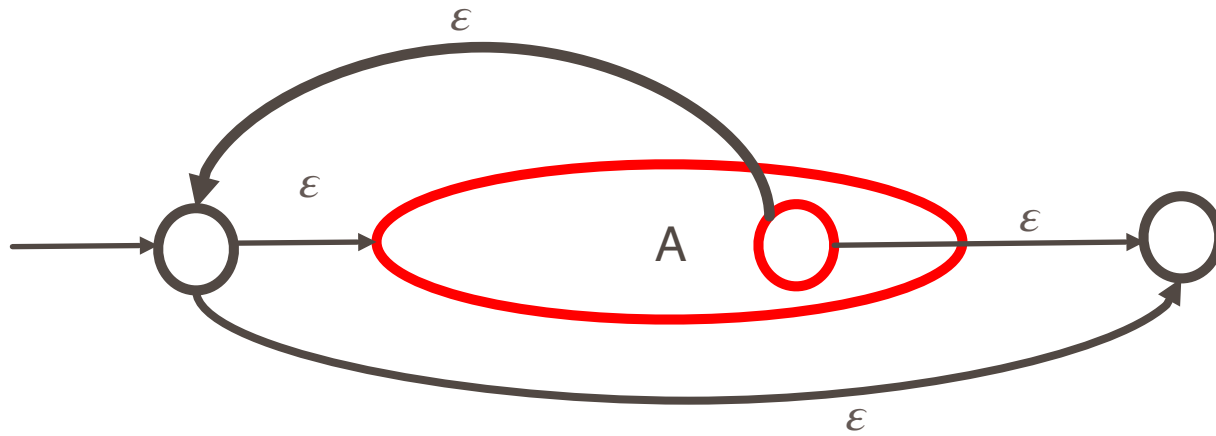


- For $A + B$



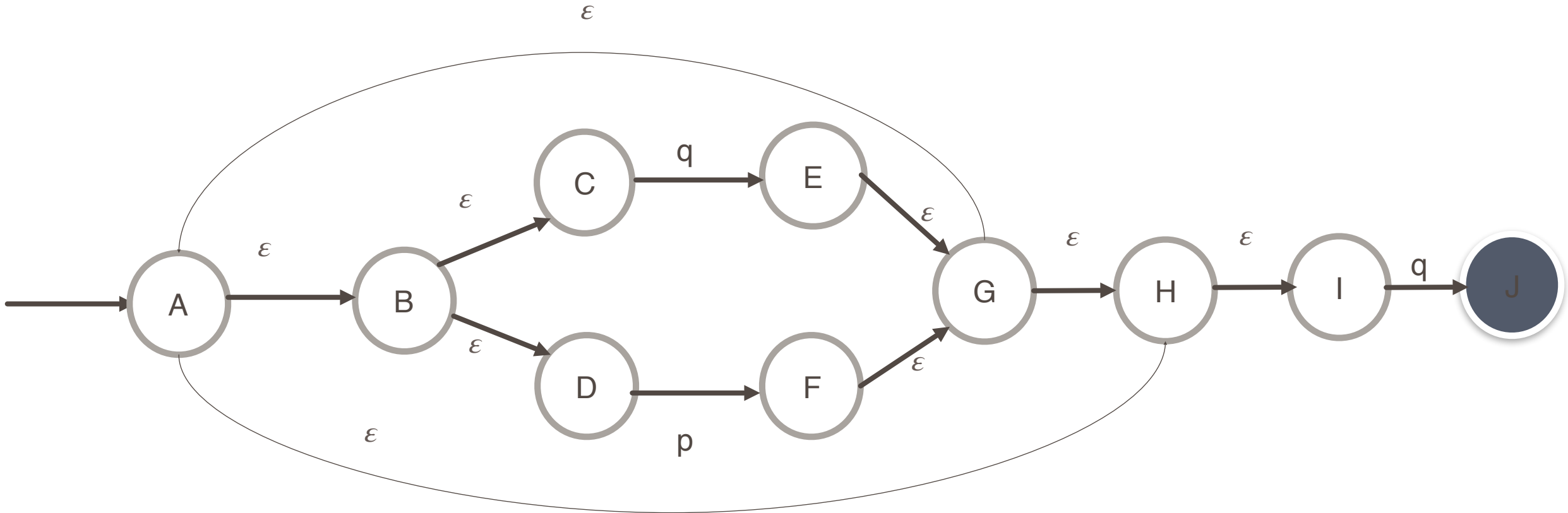
Regular Expression to NFA

- For A^*



Example

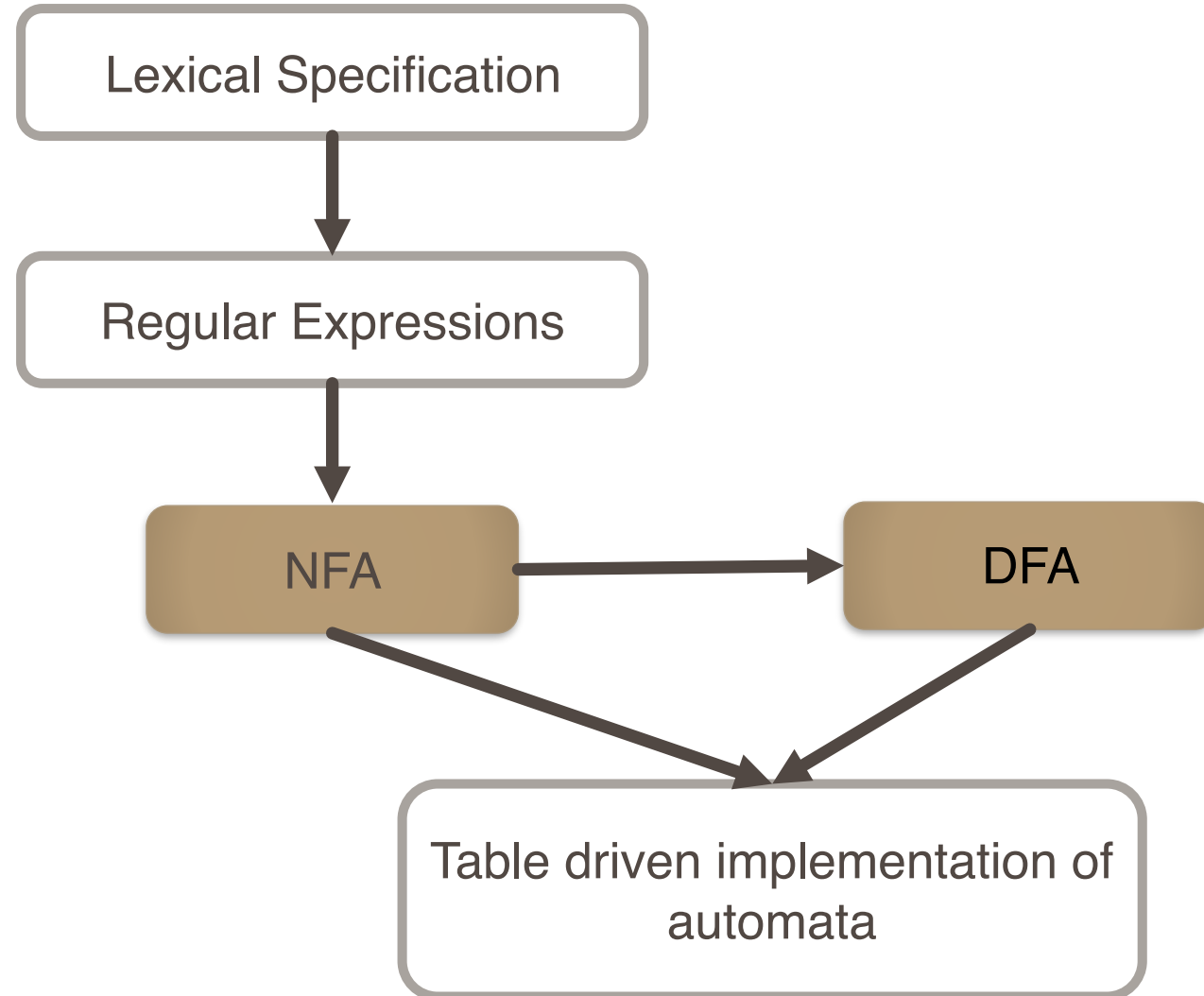
- $(q+p)^*q$



Example

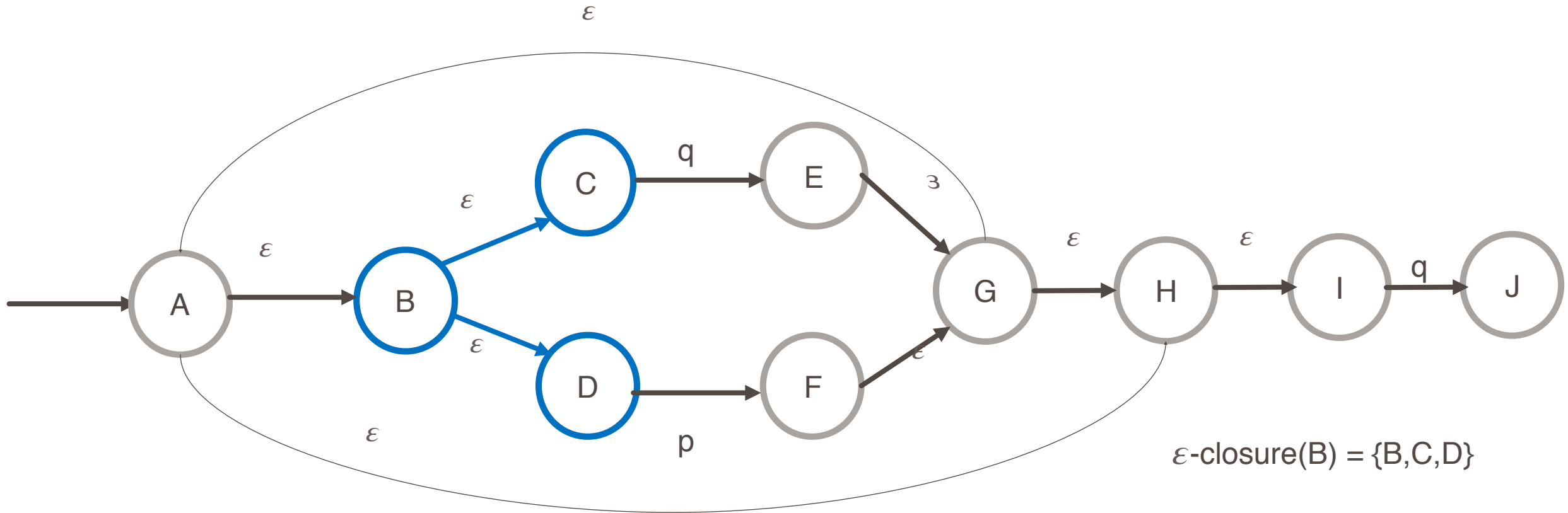
Choose the NFA that accepts the regular expression: $1^* + 0$.

NFA to DFA



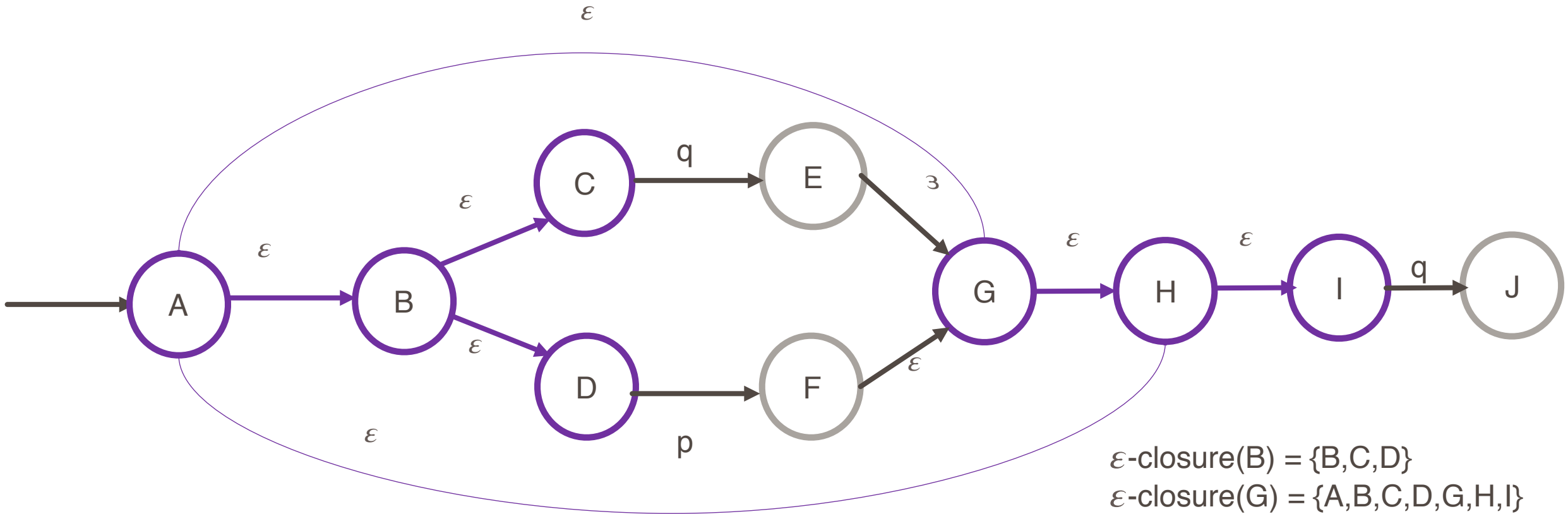
ϵ -closure

- ϵ -closure of a state is all the state I can reach following ϵ move .



ϵ -closure

- ϵ -closure of a state is all the state I can reach following ϵ move .



NFA

- An NFA can be in many states at any time
- How many different states?
 - If NFA has N states, it reaches some subset of those states, say S
 - $|S| \leq N$
 - There are $2^N - 1$ possible subsets (finite number)

NFA to DFA

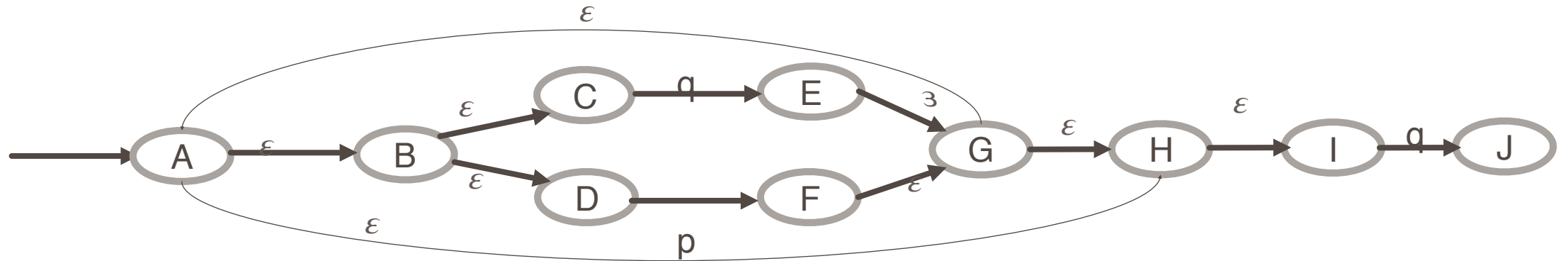
NFA

- States S
- Start s
- Final state F
- Transition state
 - $a(X) = \{y \mid x \in X \wedge x \xrightarrow{a} y\}$
- ε - closure

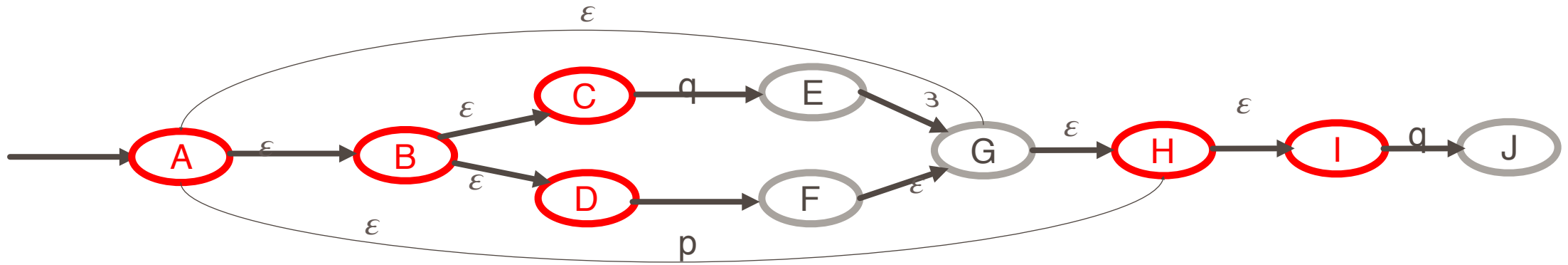
DFA

- States will be all possible subset of S except empty set
- Start state = ε - closure(s)
- Final state $\{X \mid X \cap F = \emptyset\}$
- $X \xrightarrow{a} Y$ if
 - $Y = \varepsilon$ - closure($a(X)$)

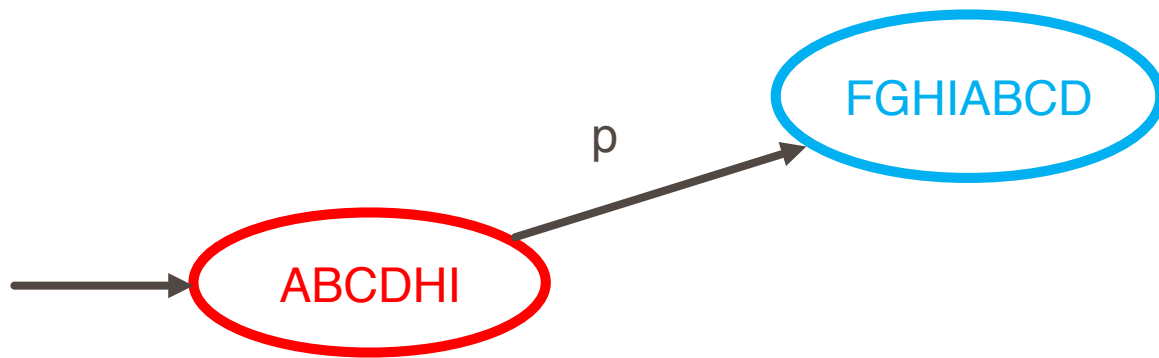
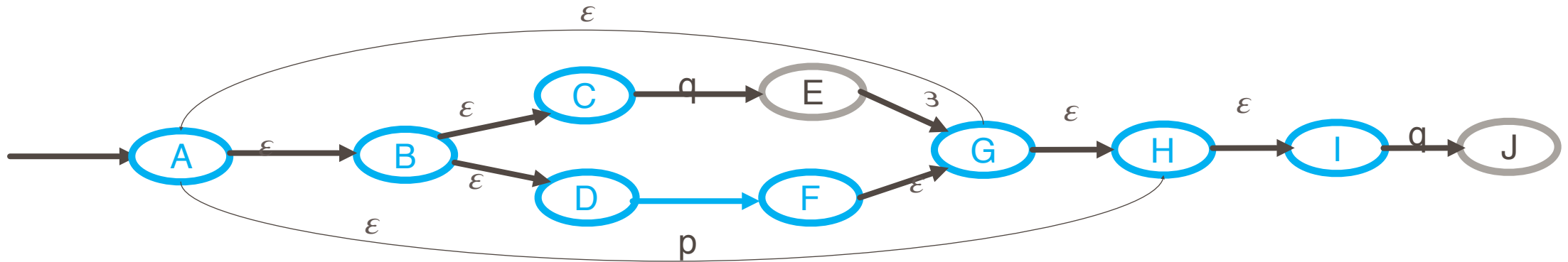
NFA to DFA



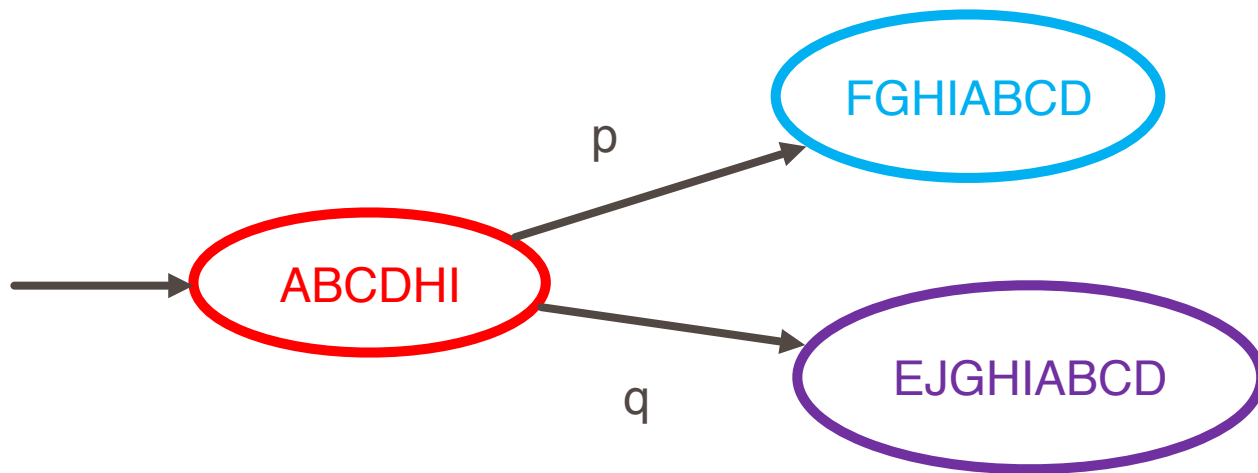
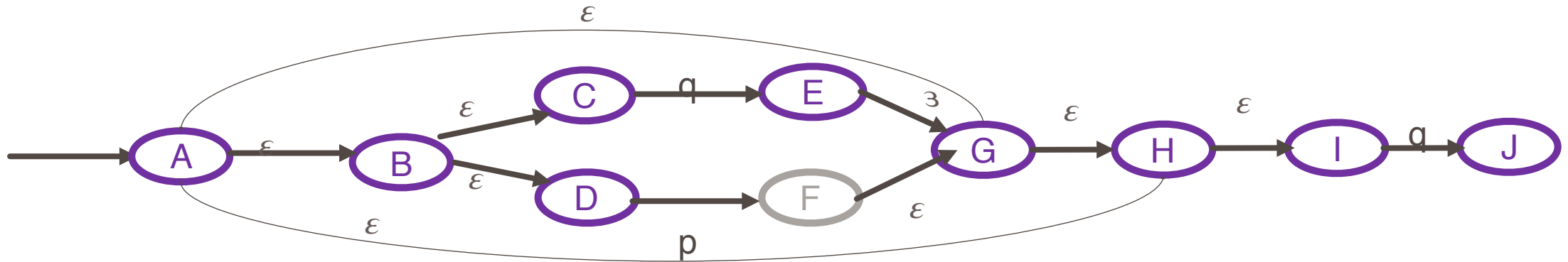
NFA to DFA



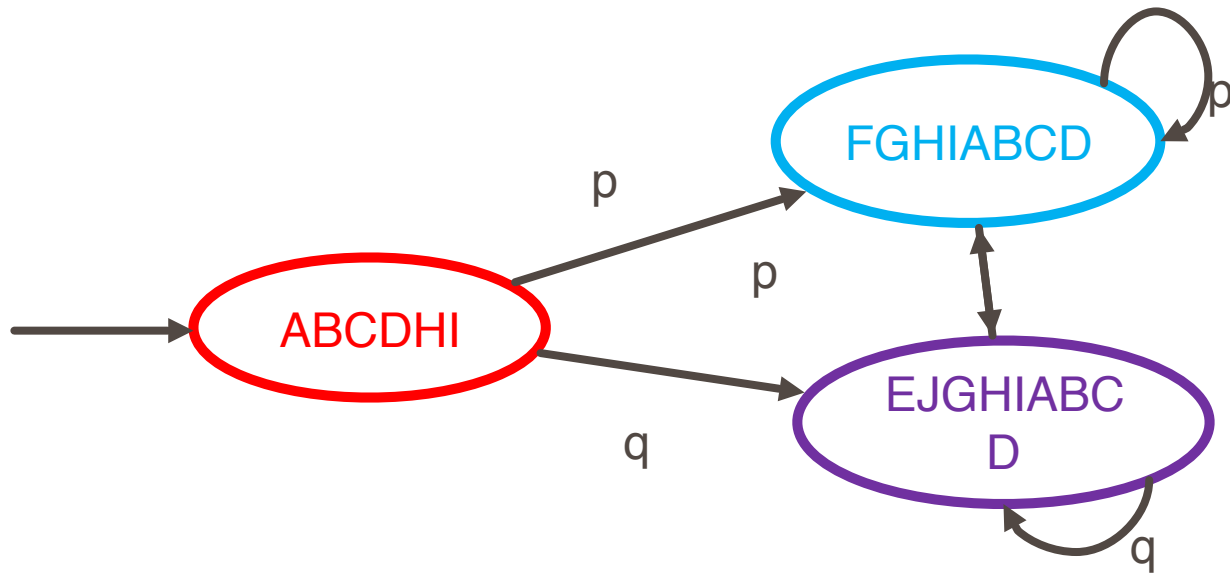
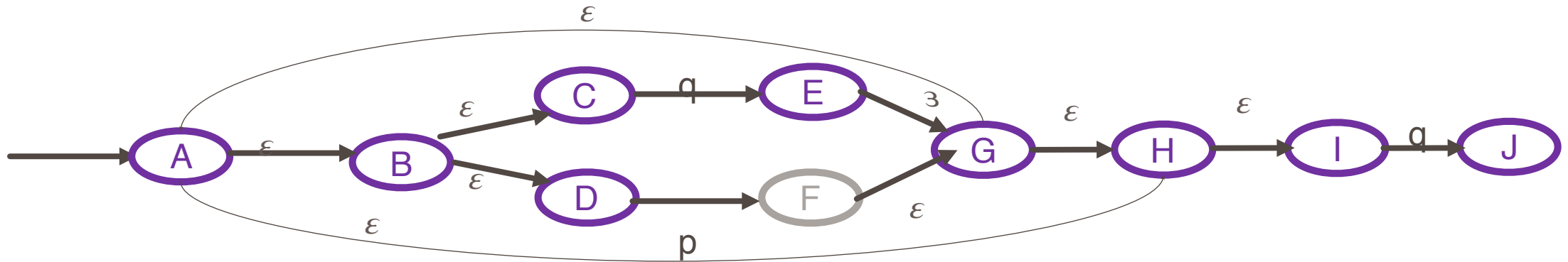
NFA to DFA



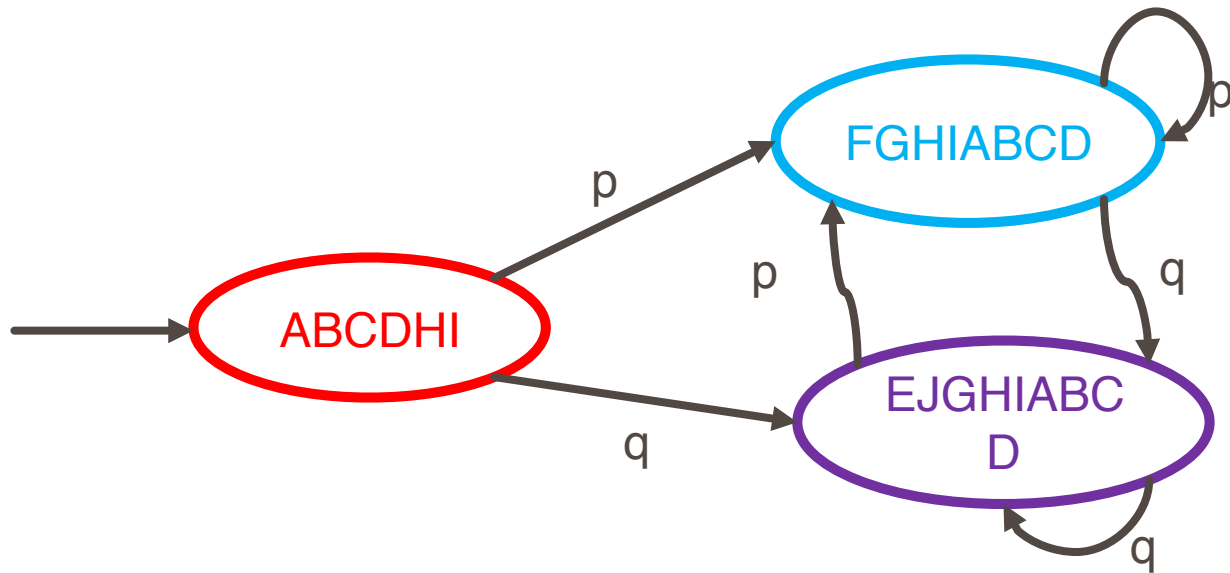
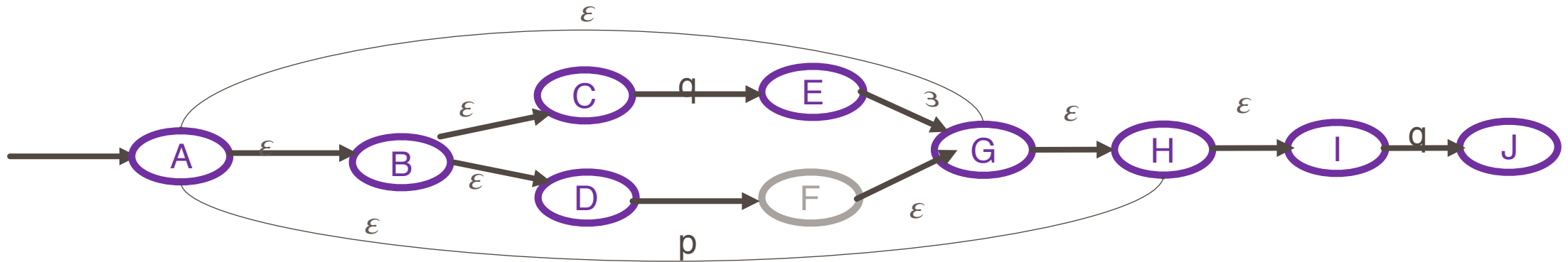
NFA to DFA



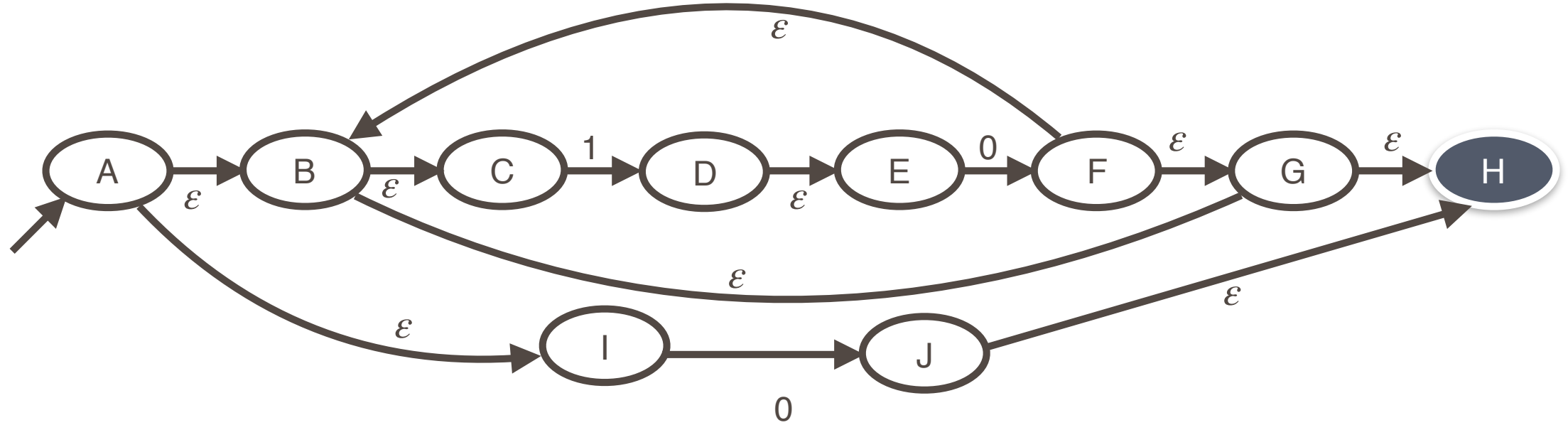
NFA to DFA



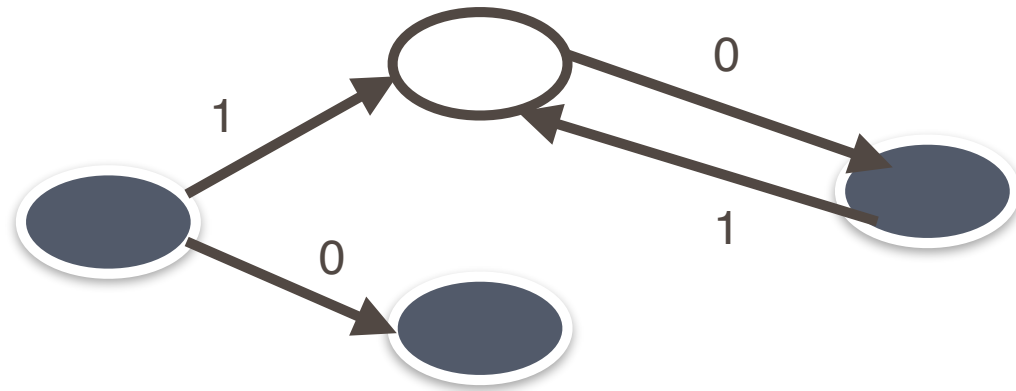
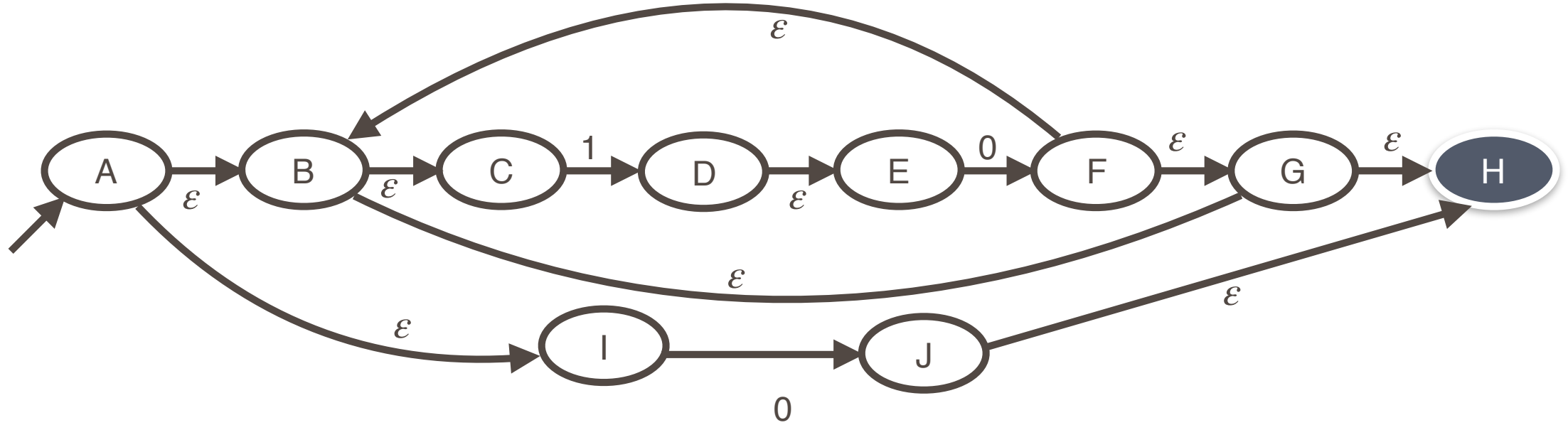
NFA to DFA



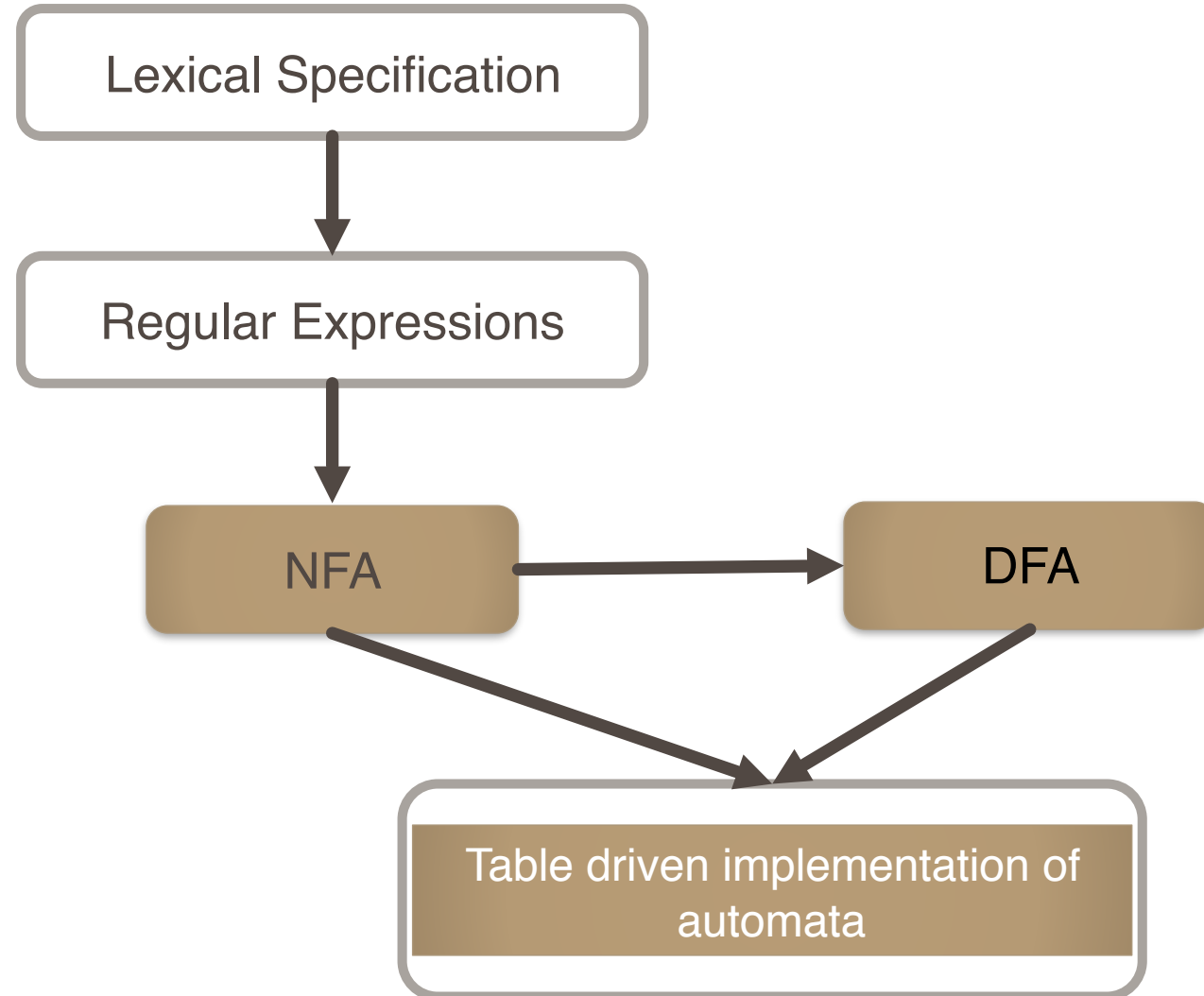
Example: NFA to DFA



Example: NFA to DFA



NFA to DFA



Implementing DFA

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

Implementing DFA

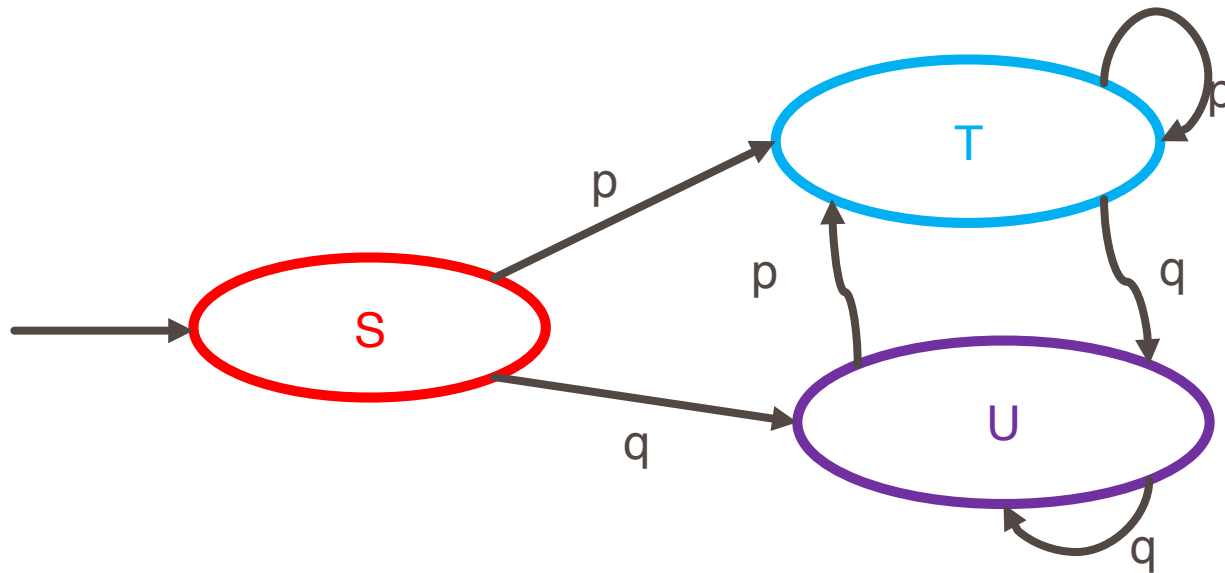


Table A

	p	q
S	T	U
T	T	U
U	T	U

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

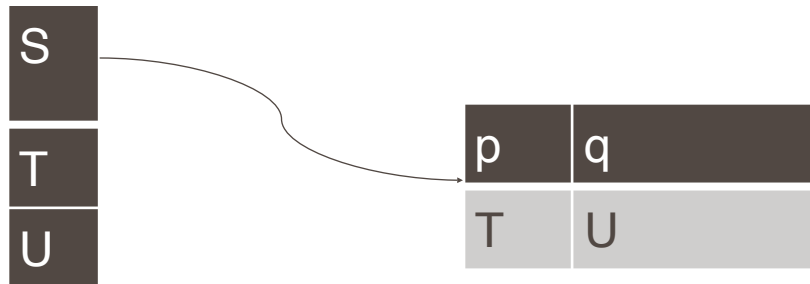
Implementing DFA

Table A

	p	q
S	T	U
T	T	U
U	T	U

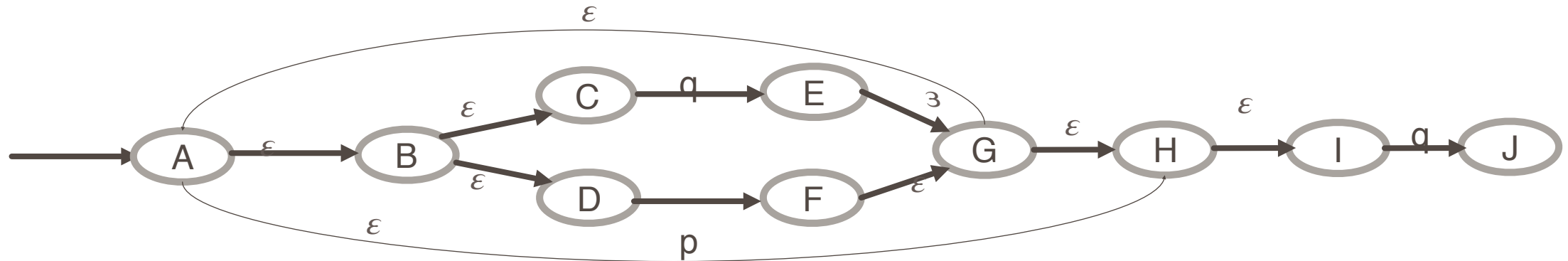
A lot of duplicate entries

Table B



Compact but need an extra indirection
- Inner loop will be slower

Implementing DFA

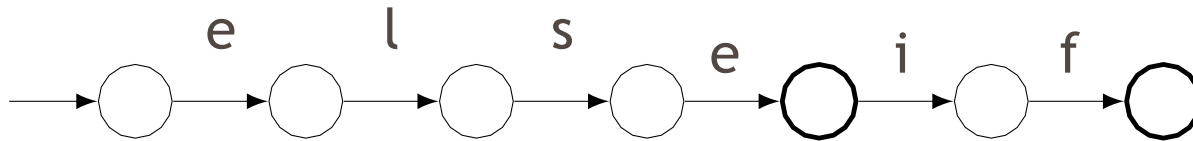


	p	q	
A			{B,H}
B			{C,D}
C		{E}	
...			

Deal with set of states rather than single state → inner loop is complicated

Deterministic Finite Automata: Example

```
{  
  type token = ELSE | ELSEIF  
}  
  
rule token =  
  parse "else"{ ELSE }  
  | "elseif"{ ELSEIF }
```



Deterministic Finite Automata

```
{ type token = IF | ID of string | NUM of string }
```

```
rule token =
```

```
  parse "if" { IF }
```

```
    | ['a'-'z'] ['a'-'z' '0'-'9'] as lit { ID(lit) }
```

```
    | ['0'-'9']+ * as num { NUM(num) }
```

