CMP4060 Languages and Compilers
Lexical Analysis – Part2

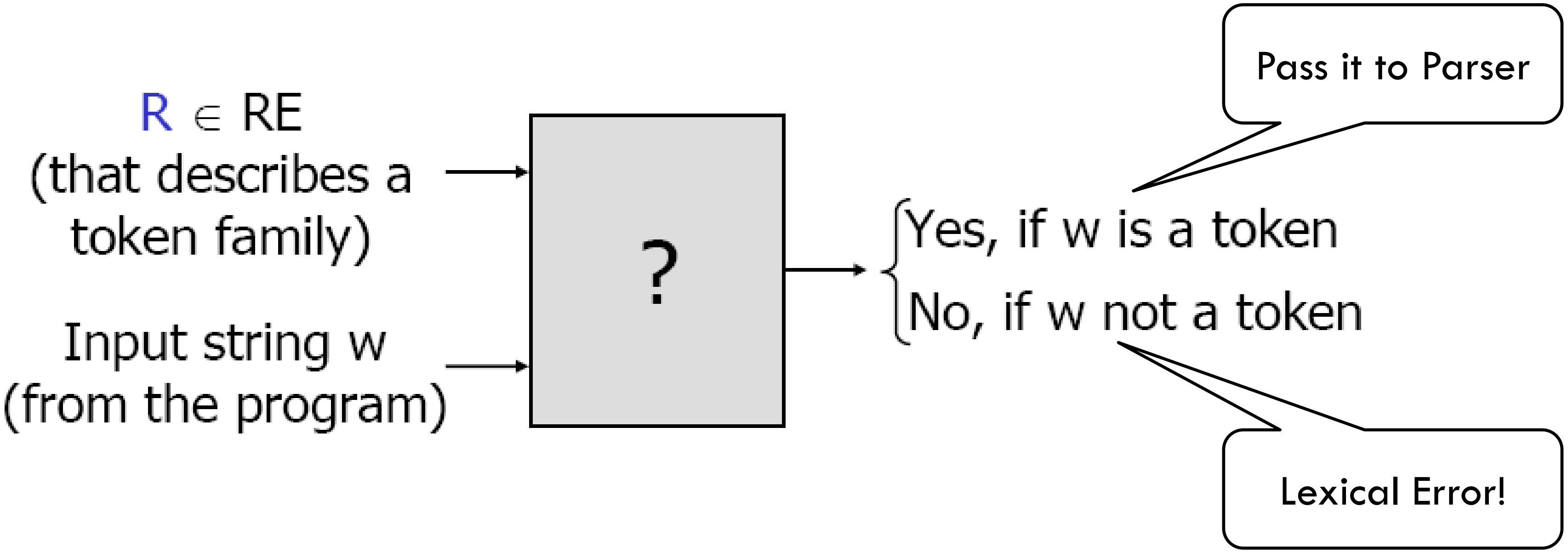
Ayman AboElHassan, PhD
Assistant Professor
ayman.abo.elmaaty@eng.cu.edu.eg

Course Outline

- Introduction to Compilers
- Lexical Analysis: Regular Grammars
- Lexical Implementation: Finite Automata
- Syntax Analysis: Context-Free Grammars
- Parser Implementation: Top-Down Parsers
- Parser Implementation: Bottom-Up Parsers
- Semantic Analysis
- Code Generation
- Code Optimization

How to use RE in Lexical Analyzer?

Given R ∈ RE and input string w, need a mechanism to determine if w ∈L(R)



Lexical Structure

Defining Lexical Structure

Step1: What is an acceptable token?

Write a regular expression for the lexemes of each token

- digit = [0-9]+
- letter = [a-zA-Z]+
- number = digit+
- identifier = letter (letter + digit + '_')*
- keyword = 'if' + 'else' + …
- openpar = '('
- …

Defining Lexical Structure

Step2: What regular expressions to use?

Construct R, matching all lexemes for all tokens

- R = keyword + identifier + number + …
  = R1 + R2 + …

Defining Lexical Structure

Step3: Check if a substring matches a regular expression?

For input string be $\mathbf{x_1 \ldots x_n}$

Check if substring $\mathbf{x_1 \ldots x_i} \in \mathbf{L(R)}$ for $i \in \{1, 2, \ldots, n\}$

Lexical Implementation performs the check

If success:

- Mark substring $\mathbf{x_1 \ldots x_i} \in \mathbf{L(R)}$
- Remove $\mathbf{x_1 \ldots x_i}$ from input string
- Repeat the operation on the remaining string

Defining Lexical Structure

Lexical implementation needs to handle ambiguities

Token Length

- If 2 substrings $\mathbf{x_1 \ldots x_i}$ & $\mathbf{x_1 \ldots x_k}$ match the same regex **L(R)**
- Solution: Pick longest possible substring matching

Which Token

- If the same substring $\mathbf{x_1 \ldots x_i}$ matches 2 regex $\mathbf{L(R_m)}$ & $\mathbf{L(R_n)}$
- Solution: Pick the first regex in the list

Finite Automata

Finite Automata

Regular Expression → Specification “Rules”

Finite Automata → Implementation

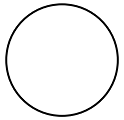
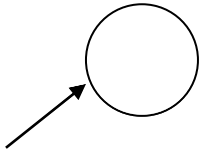
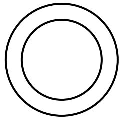
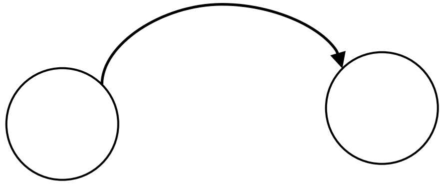
Finite Automata

A finite automaton consists of

- An input alphabet $\sum$
- A set of states S
- A start state n
- A set of accepting states $\mathrm{F} \subseteq \mathrm{S}$
- A set of transitions: $state \xrightarrow{input} state$

Finite Automata (State Graph)

A finite automaton consists of

- An input alphabet $\sum$
- A set of states S ➔ 
- A start state n ➔ 
- A set of accepting states $\mathrm{F} \subseteq \mathrm{S}$ ➔ 
- A set of transitions: $state \xrightarrow{input} state$ ➔ 

Finite Automata (State Graph)

Example

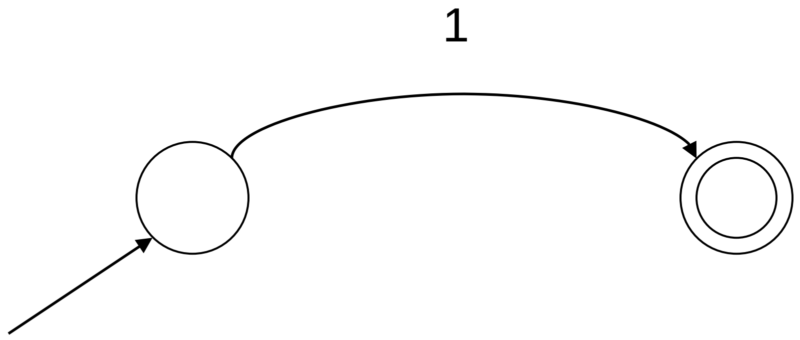
A finite automaton that accepts only "1"

Finite Automata (State Graph)

Example

A finite automaton that accepts only “1”

- L(R) = ‘1’



Finite Automata (State Graph)

Example

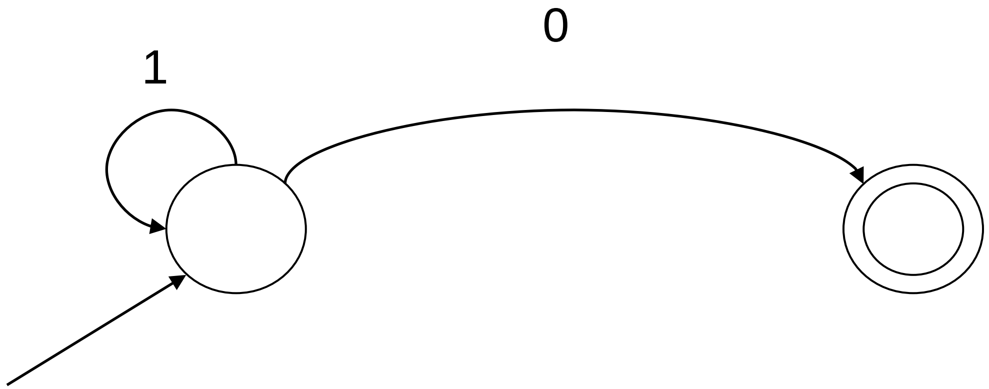
A finite automaton that any number of 1s followed by a single 0

Finite Automata (State Graph)

Example

A finite automaton that any number of 1s followed by a single 0

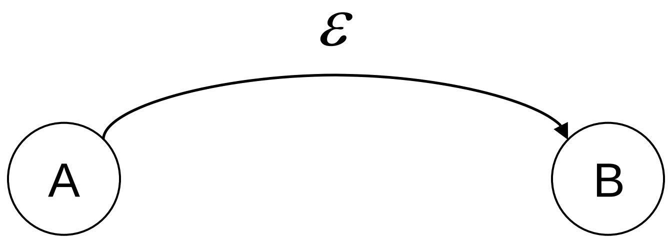
- L(R) = (1)*0



Finite Automata (State Graph)

Epsilon Moves

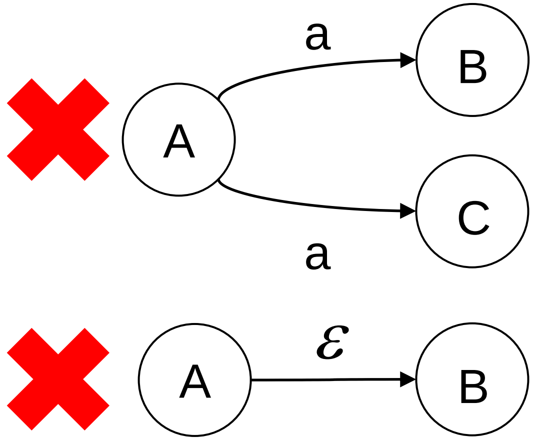
- Another kind of transition
- Machine can move from state A to state B without reading input



Finite Automata (State Graph)

Deterministic Finite Automata (DFA)

- One transition per input per state
- No epsilon-moves



Nondeterministic Finite Automata (NFA)

- Can have multiple transitions for one input in a given state
- Can have epsilon-moves

NFAs & DFAs recognize the same set of languages (regular languages)

NFA vs DFA

DFA can take only one path through the state graph

- Completely determined by input
- Faster to execute (There are no choices to consider)

NFAs can choose

- Whether to make epsilon-moves
- Which of multiple transitions for a single input to take
- Rule: NFA accepts if it <u>can</u> get to a final state

NFA vs DFA

For a given language NFA can be simpler than DFA

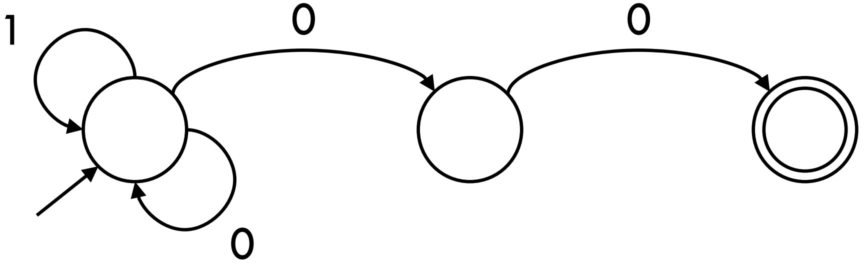
- Example: $L(R) = (1|0)^*00$

NFA vs DFA

For a given language NFA can be simpler than DFA

- Example: L(R) = (1|0)*00

NFA

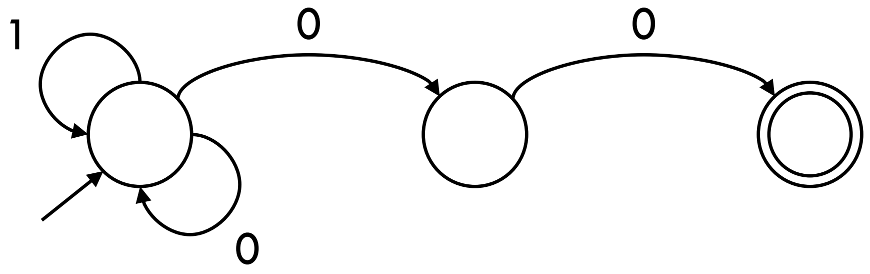


NFA vs DFA

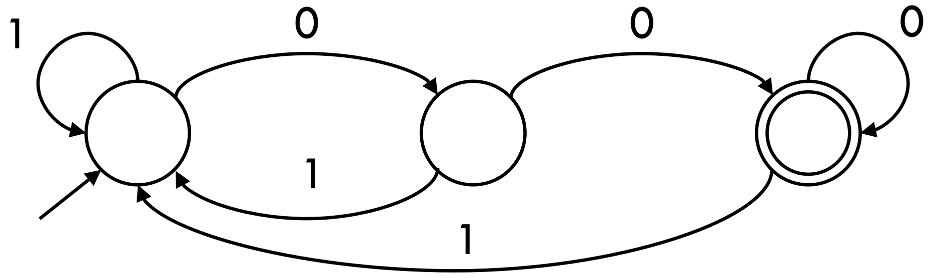
For a given language NFA can be simpler than DFA

- Example: L(R) = (1 | 0)*00

NFA



DFA



NFA vs DFA

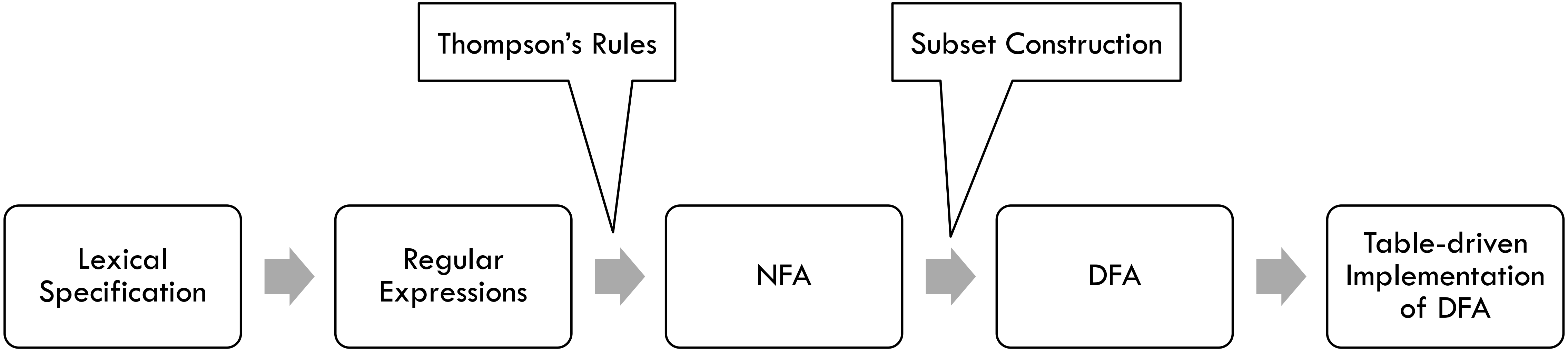
For a given language NFA can be simpler than DFA

DFA can be exponentially larger than NFA

- It is possible to convert an NFA to a DFA

Automaton	Initial	Per String
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA typical case	$O(\|r\|^3)$	$O(\|x\|)$
DFA worst case	$O(\|r\|^2 2^{\|r\|})$	$O(\|x\|)$

From Regular Expressions to Finite Automata

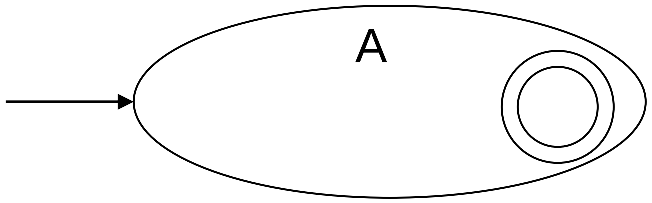


Thompson's Rules

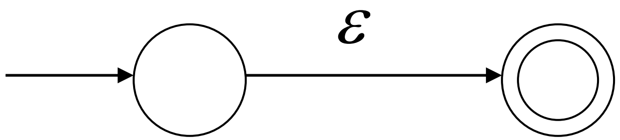
Thompson’s Rules

For each kind of regular expression, define an NFA

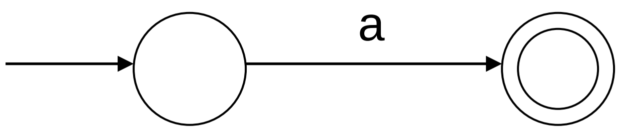
For A



For ε



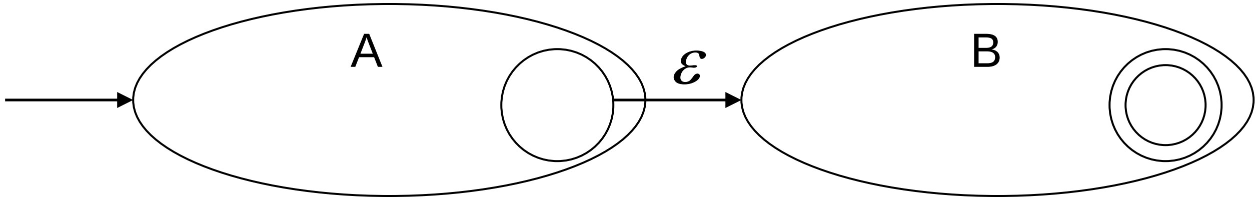
For input a



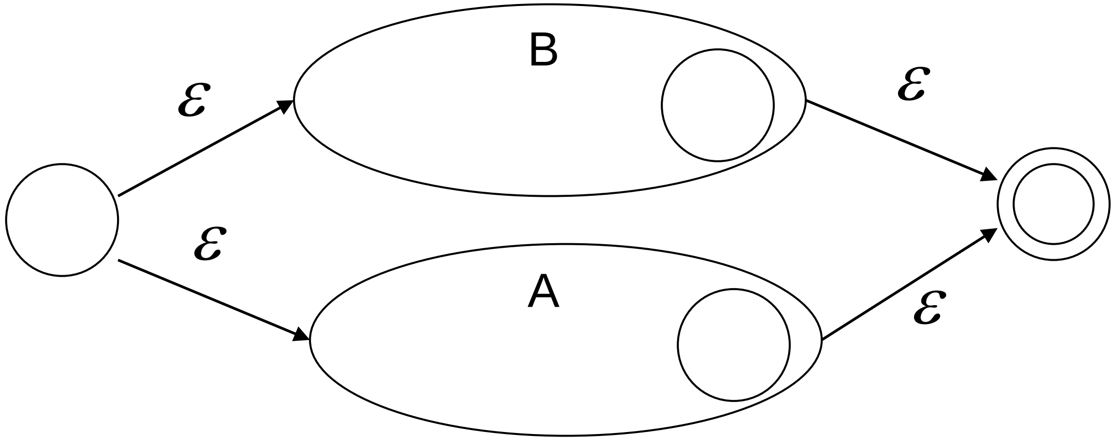
Thompson's Rules

For each kind of regular expression, define an NFA

For AB



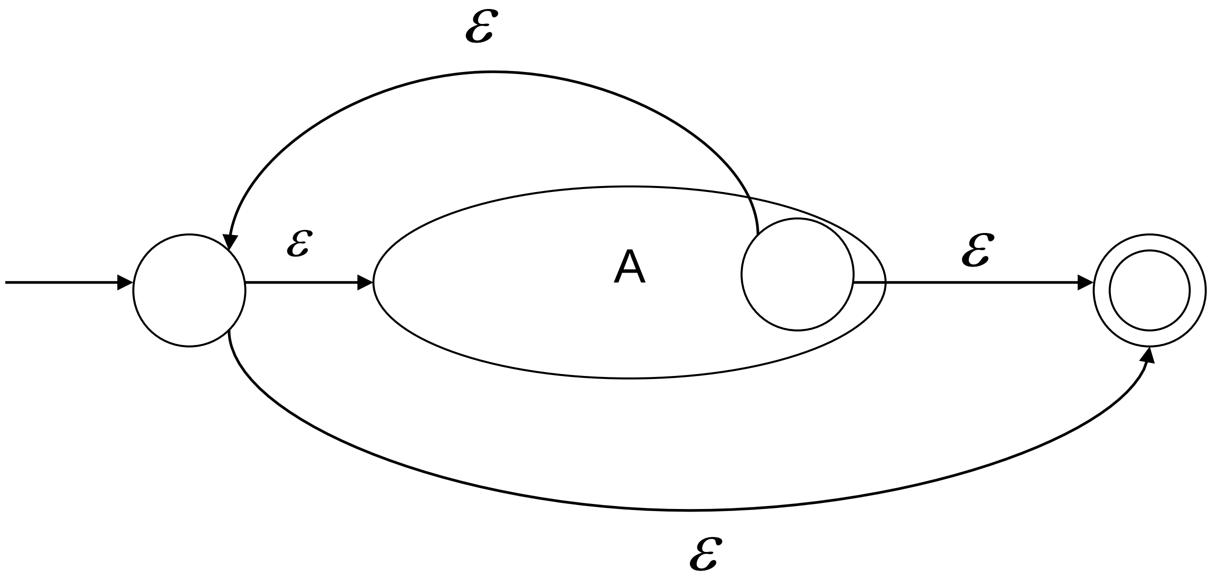
For A+B



Thompson's Rules

For each kind of regular expression, define an NFA

For A*



Thompson's Rules

Example: (1+0)*1

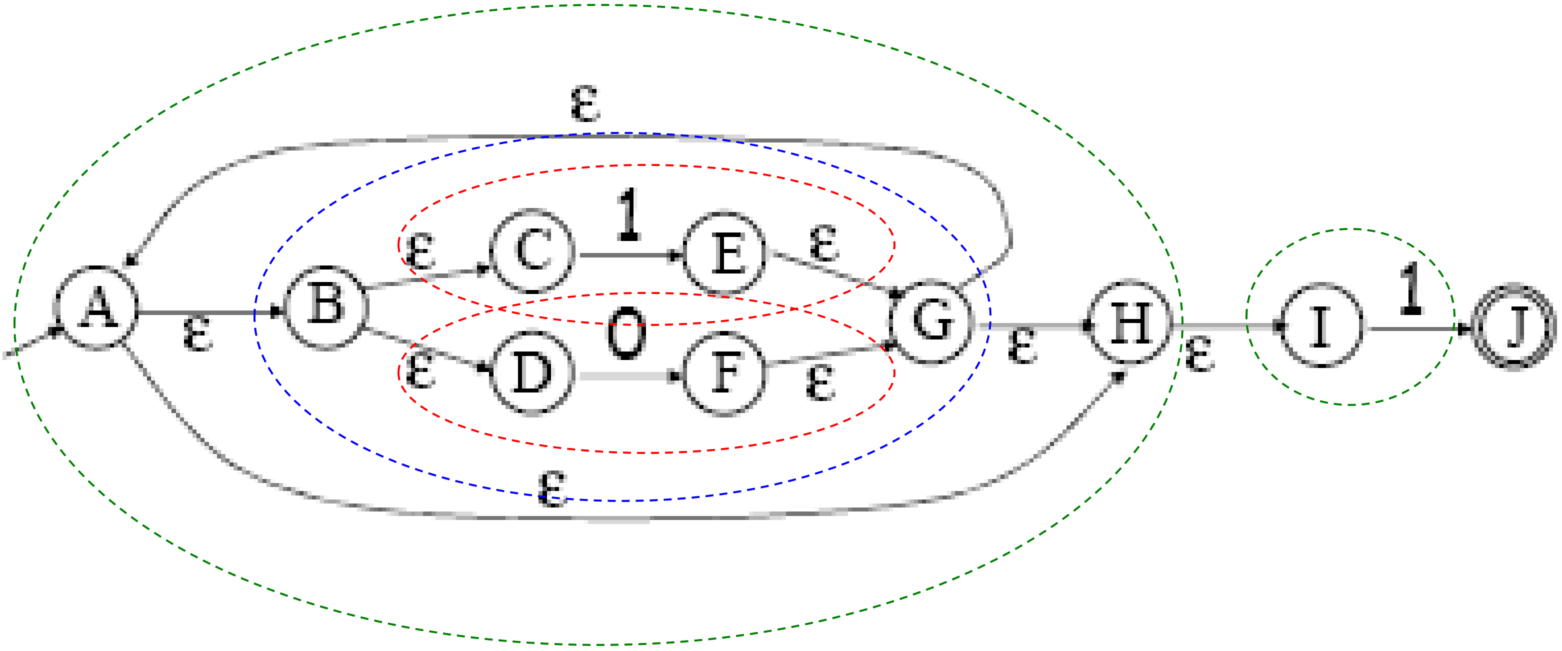
Thompson's Rules

Example: (1+0)*1

$$\underbrace{(\underbrace{\underbrace{1}_{M_1} + \underbrace{0}_{M_2}}_{M_4})^*}_{M_5}\ \underbrace{1}_{M_3}$$

Thompson's Rules

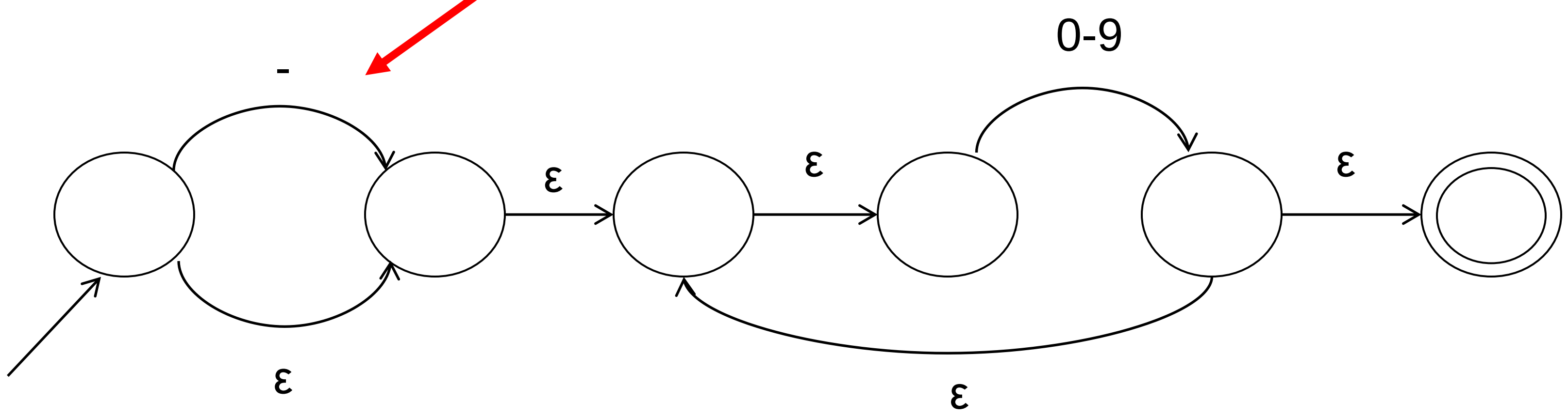
Example: (1+0)*1



Thompson’s Rules

Example: -?[0-9]+

$(\text{‘-’} + \varepsilon)\ [0\text{-}9]+$



Subset Construction Algorithm

Subset Construction Algorithm

Each state of DFA (D) is a set of NFA (N) states

Simulate "in parallel" all possible moves NFA can make on a given input string

- s is a single state of N
- T is a set of states of N

OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}\ \epsilon\text{-}closure(s)$.
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some state s in T.

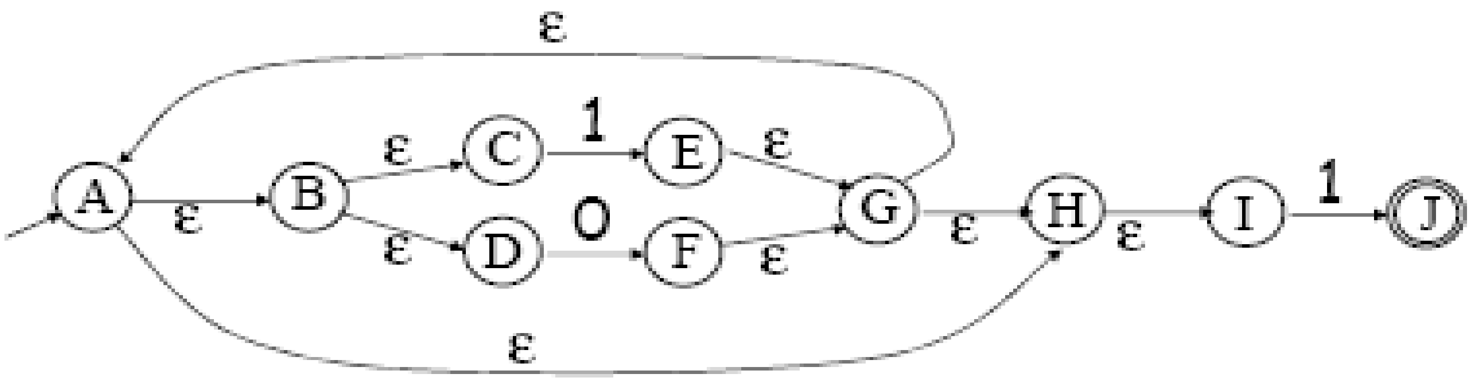
Subset Construction Algorithm

Each state of DFA → a non-empty subset of states of the NFA

1. Start state = ε-closure(S_0)
   - S_0 is its start state
   - The set of NFA states reachable through ε-moves from NFA start state
2. T' = ε-closure(move(T,a))
   - T' is the set of NFA states reachable from any state in T after reading the input a, considering ε-moves as well
3. Add a transition $\mathrm{T} \xrightarrow{a} \mathrm{T}'$

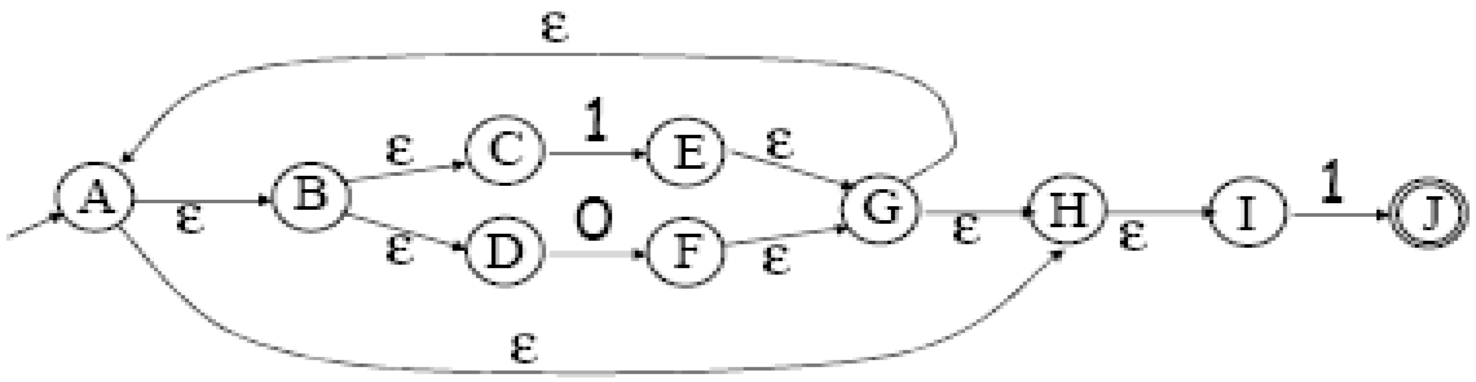
Subset Construction Algorithm

Example: (1+0)*1



Subset Construction Algorithm

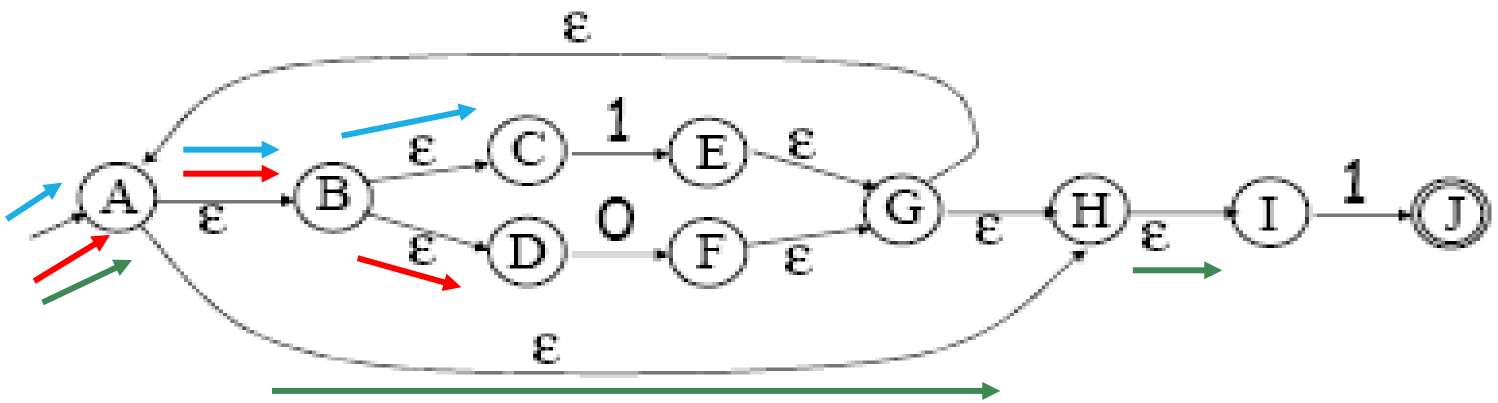
Example: (1+0)*1



Step1: Get start state “reachable through ε-moves”

Subset Construction Algorithm

Example: (1+0)*1

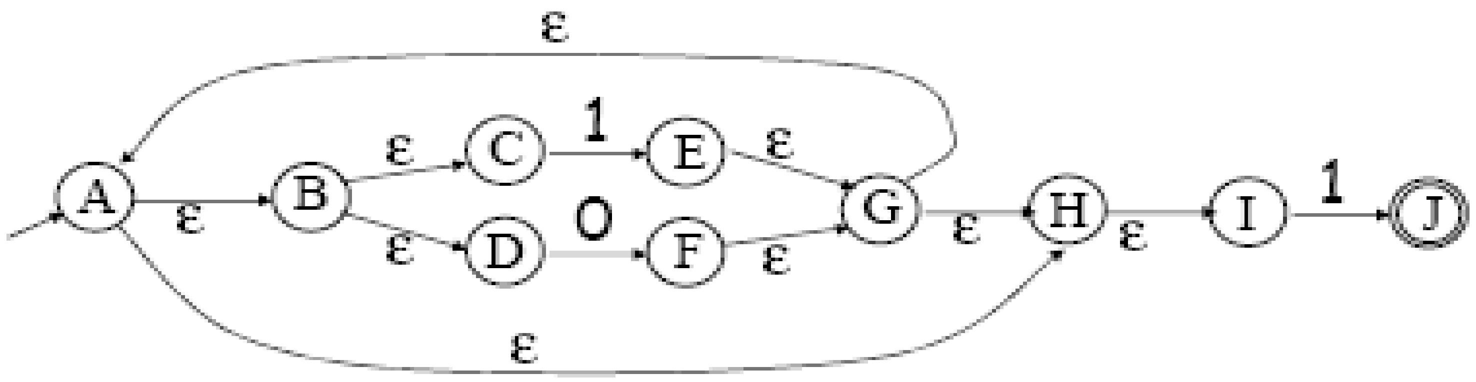


Step1: Get start state “reachable through ε-moves”

ABCDHI

Subset Construction Algorithm

Example: (1+0)*1

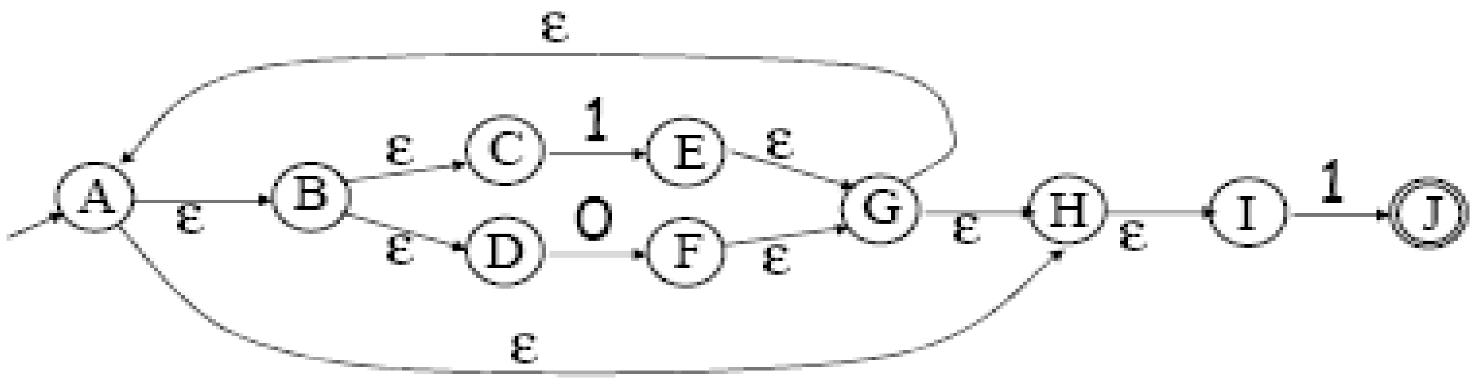


Step2: States T' reachable from T by a (& ε)

ABCDHI

Subset Construction Algorithm

Example: (1+0)*1



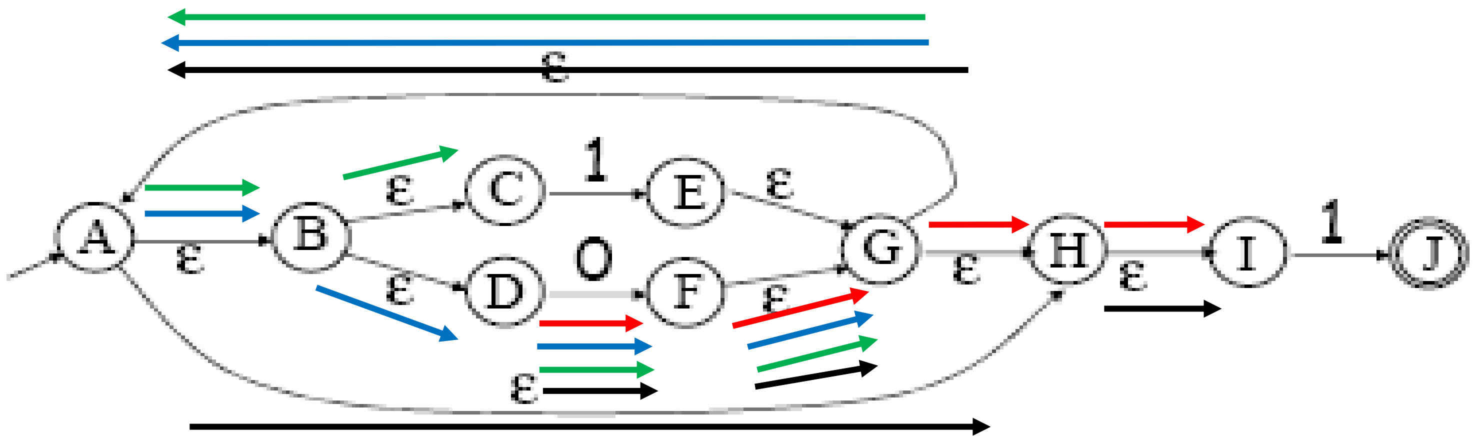
Step2: States T' reachable from T by a (& ε)

- T = {A,B,C,D,H,I}
- a = 0

ABCDHI

Subset Construction Algorithm

Example: (1+0)*1



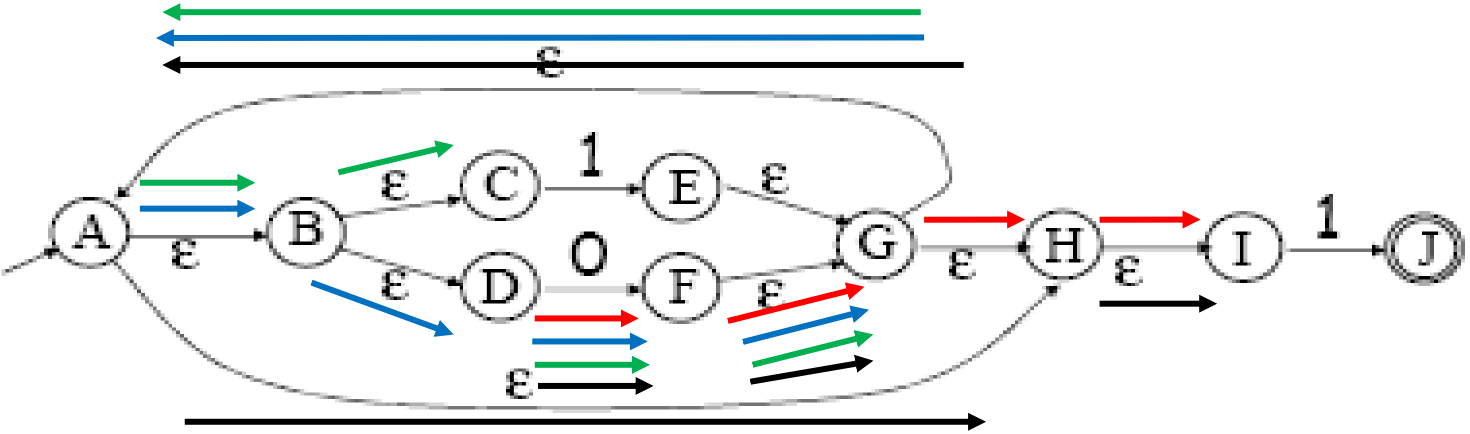
Step2: States T' reachable from T by a (& ε)

- T = {A,B,C,D,H,I}
- a = 0

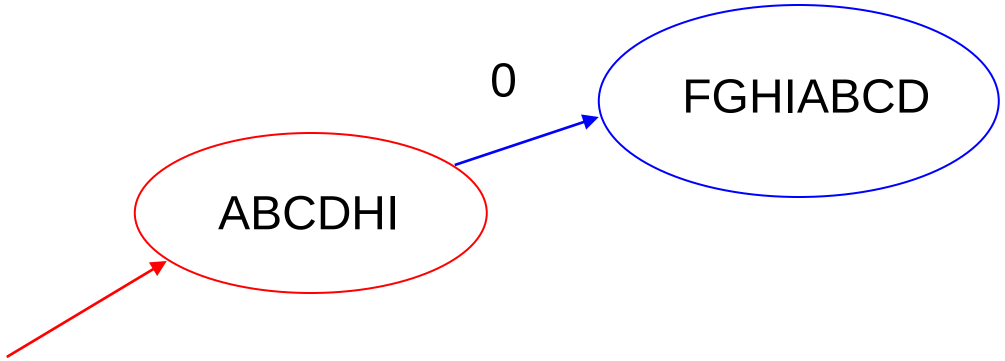
ABCDHI

Subset Construction Algorithm

Example: (1+0)*1

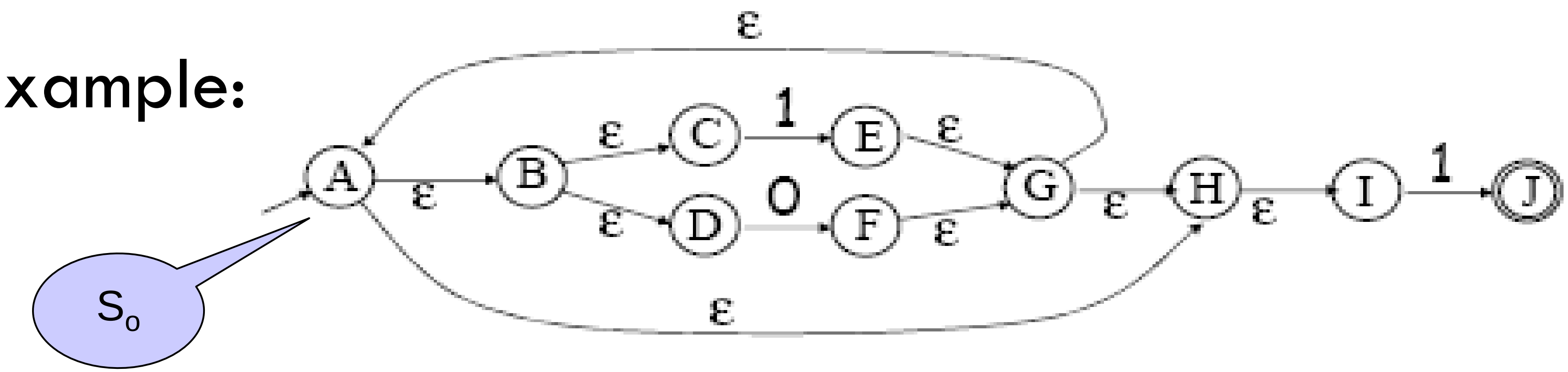


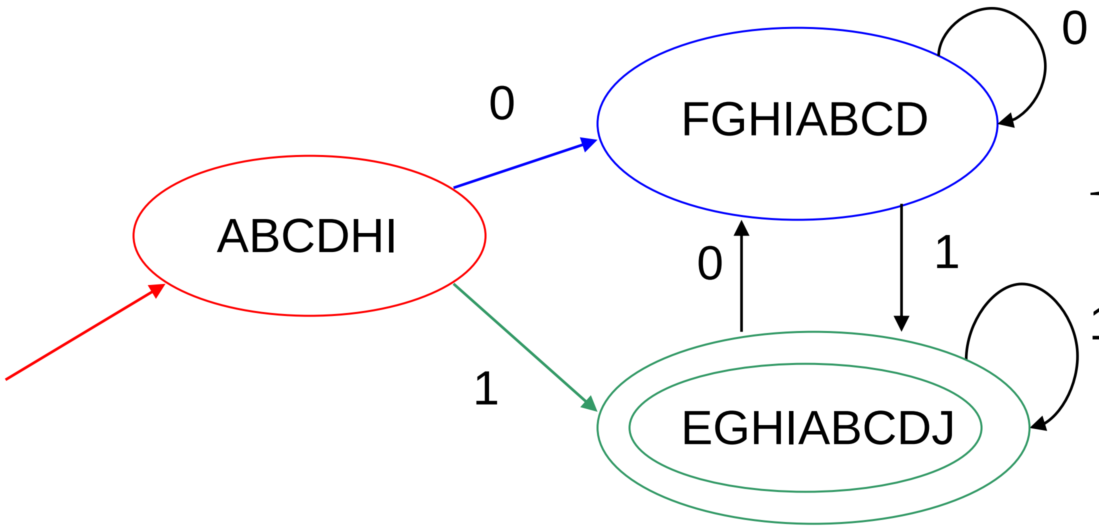
Step3: Add a transition $T \xrightarrow{a} T'$



Subset Construction Algorithm

Example:

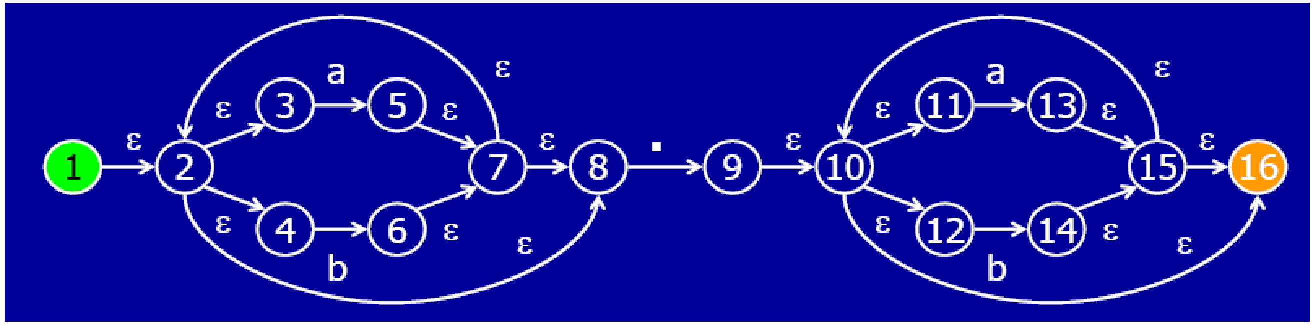




<u>DFA Graph:</u>
- No epsilon moves
- Single path per input

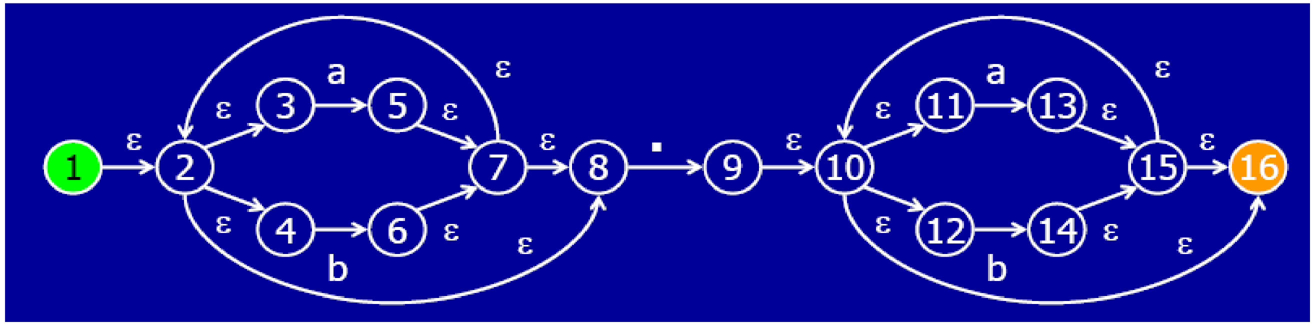
Subset Construction Algorithm

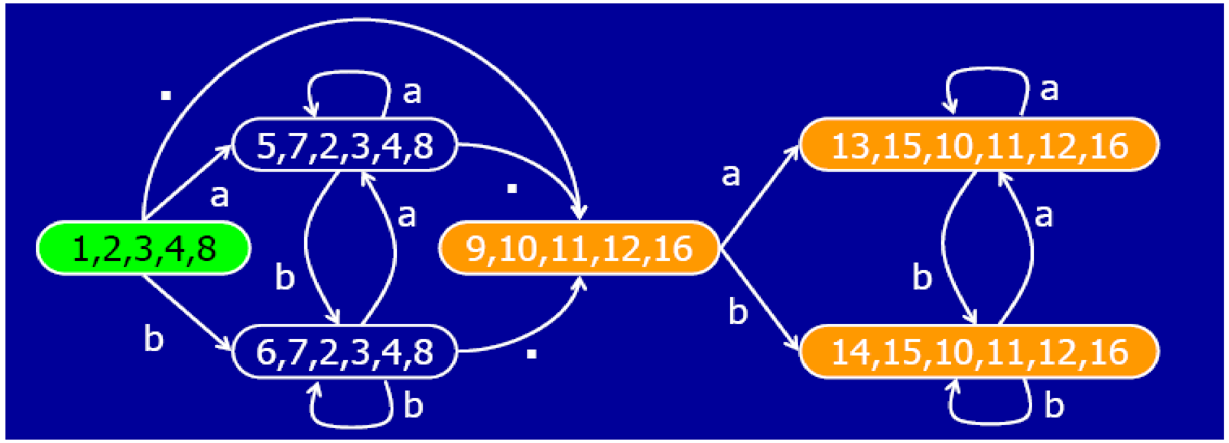
Example (a+b)*.(a+b)



Subset Construction Algorithm

Example (a+b)*.(a+b)





DFA Table Implementation

DFA Table Implementation

A DFA can be implemented by a 2D table T

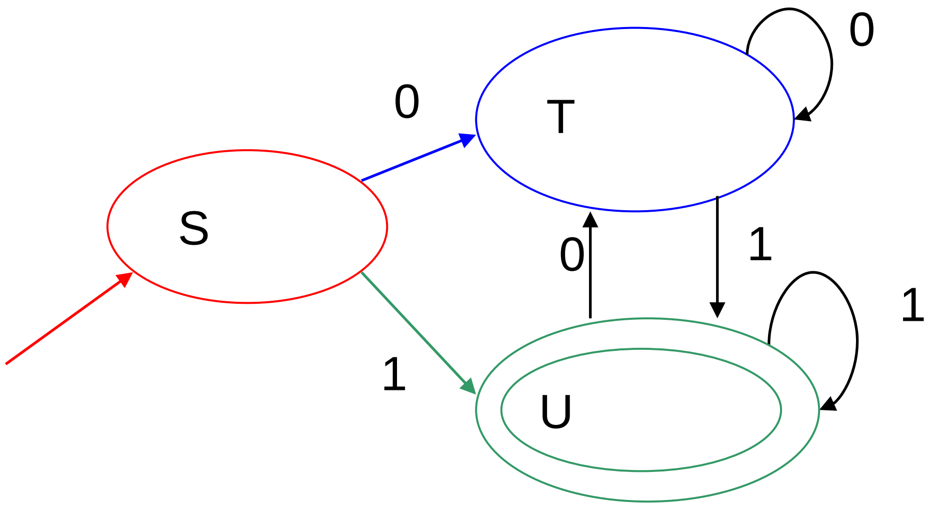
- One dimension is **states**
- Other dimension is **input symbol**
- For every transition $S_i \xrightarrow{a} S_k$ define T[i,a] = k

DFA execution

- If in state S_i and input a, read T[i,a] = k and skip to state S_k
- Very efficient

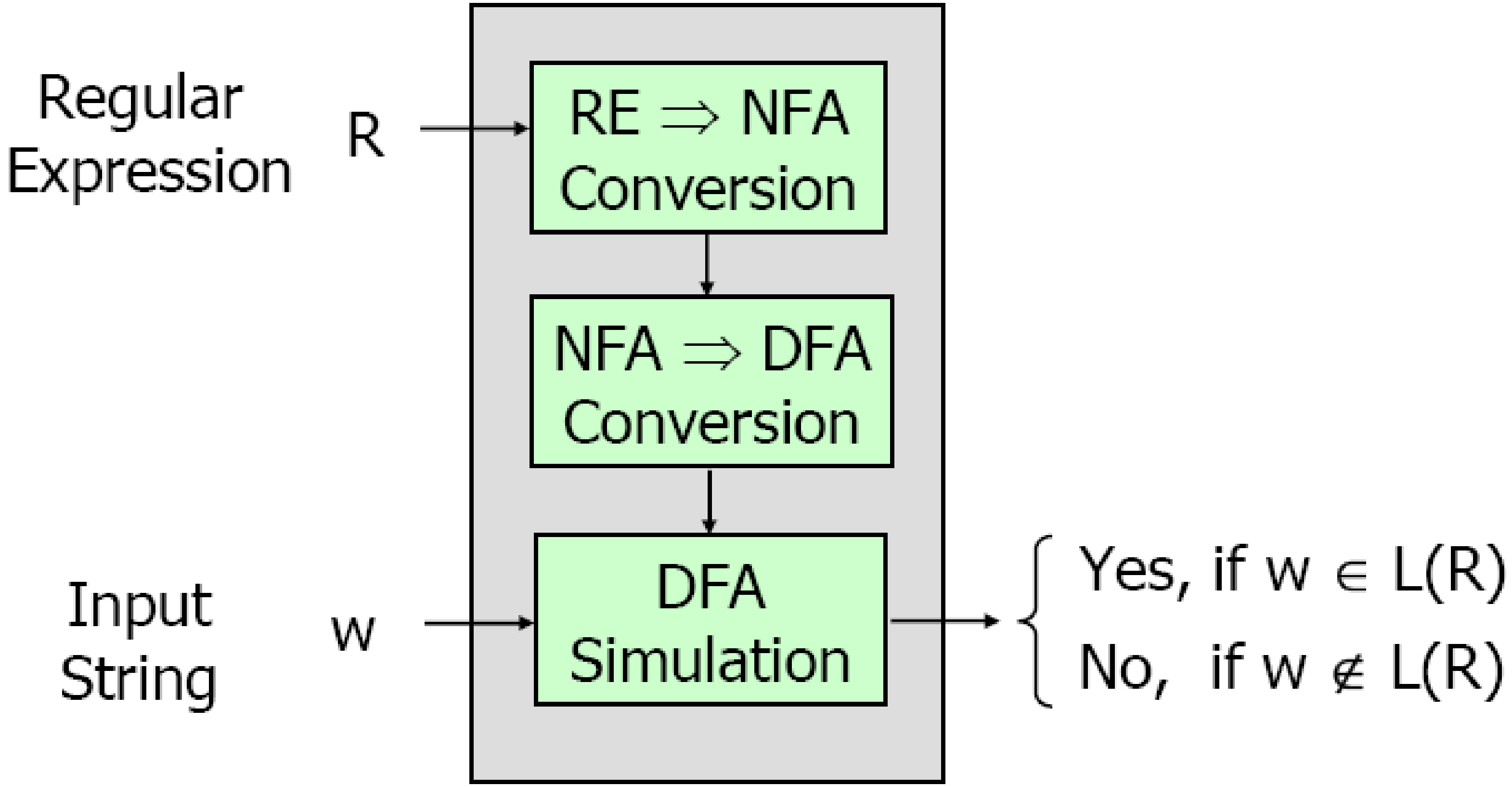
DFA Table Implementation

Example



	0	1
S	T	U
T	T	U
U	T	U

Putting the Pieces Together

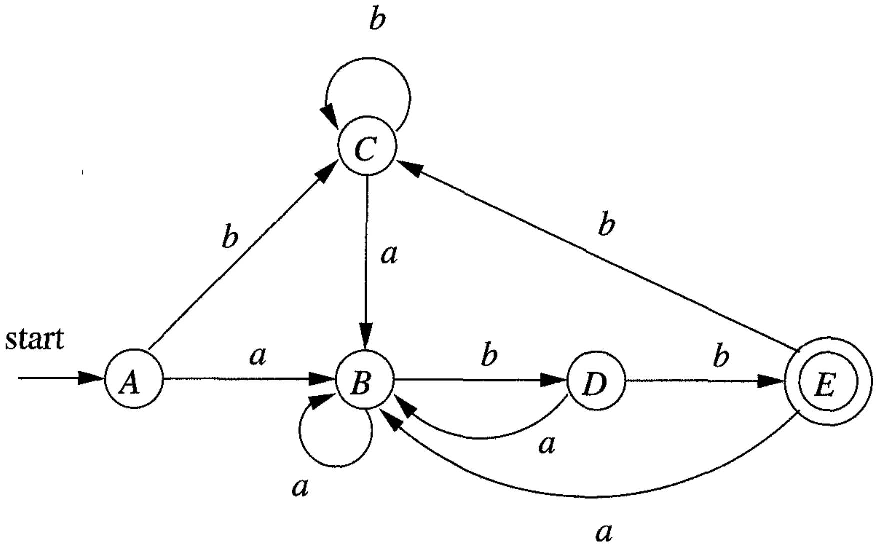


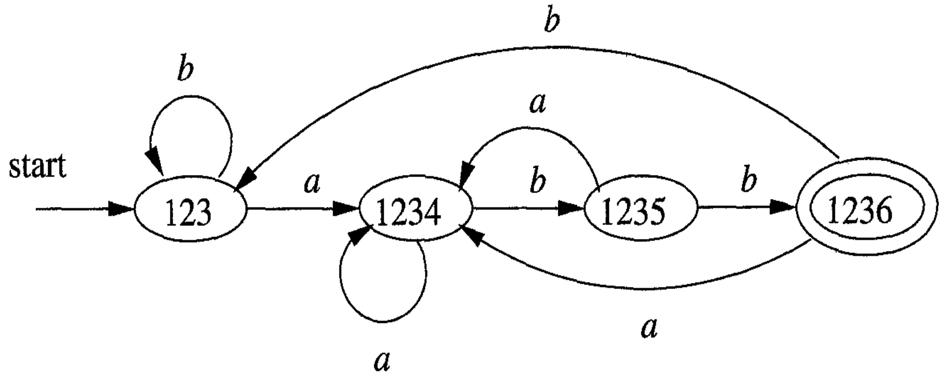
DFA Minimization

DFA Minimization

Multiple DFAs can be used to recognize the same language

Example: (a+b)*abb





DFA Minimization

These automata don't have the same number of states

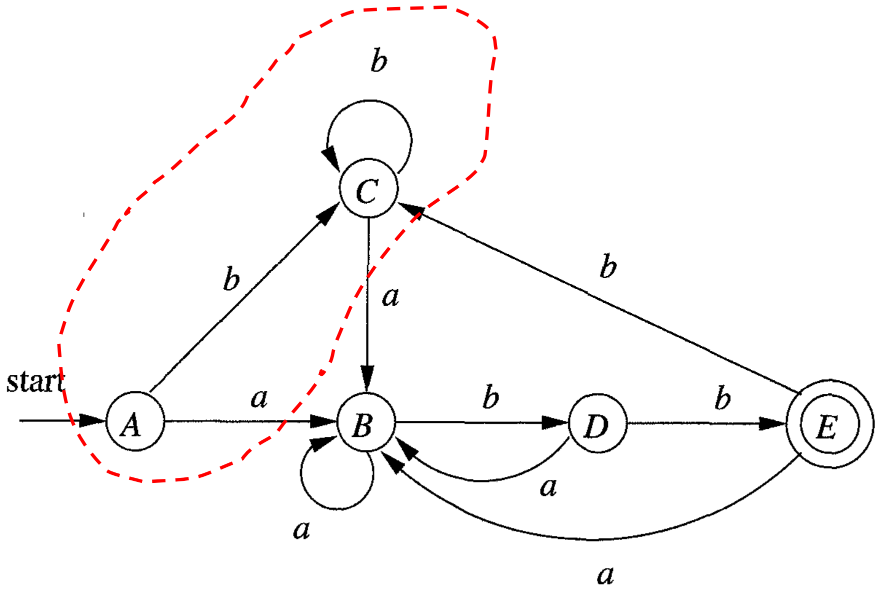
When implementing a lexical analyzer as a DFA

- Generally, prefer a DFA with as few states as possible
- Each state requires an entry in the DFA table implementation

DFA Minimization

State A and C are equivalent

- On any input they transfer to the same state
  - $A \xrightarrow{a} B, C \xrightarrow{a} B$
  - $A \xrightarrow{b} C, C \xrightarrow{b} C$

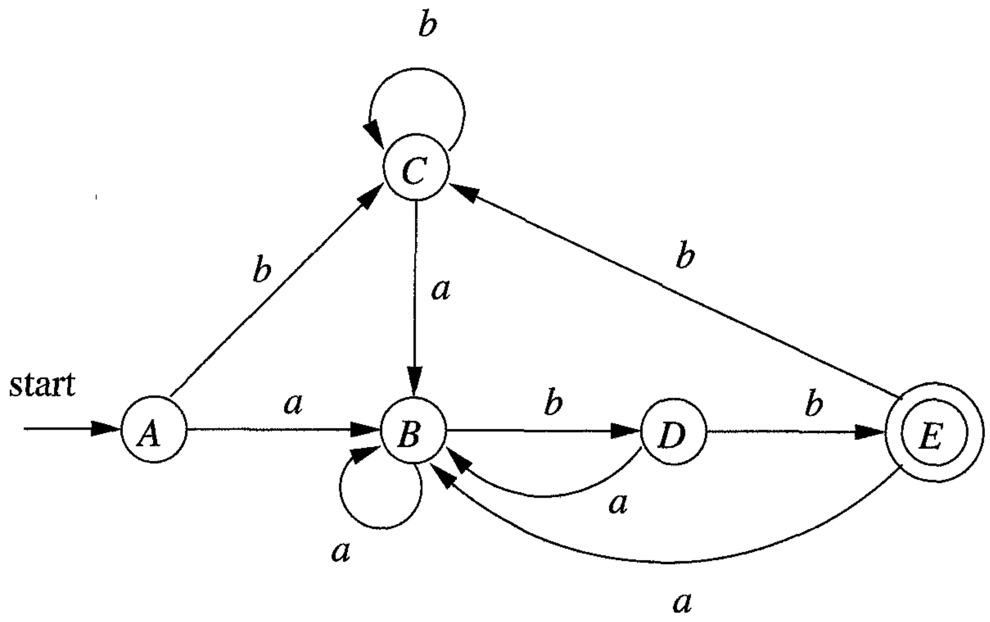


DFA Minimization

Step1: Partition graph π into 2 groups (accepting, non-accepting)

Example

- {A, B, C, D} → non-accepting state
- {E} → accepting state



DFA Minimization

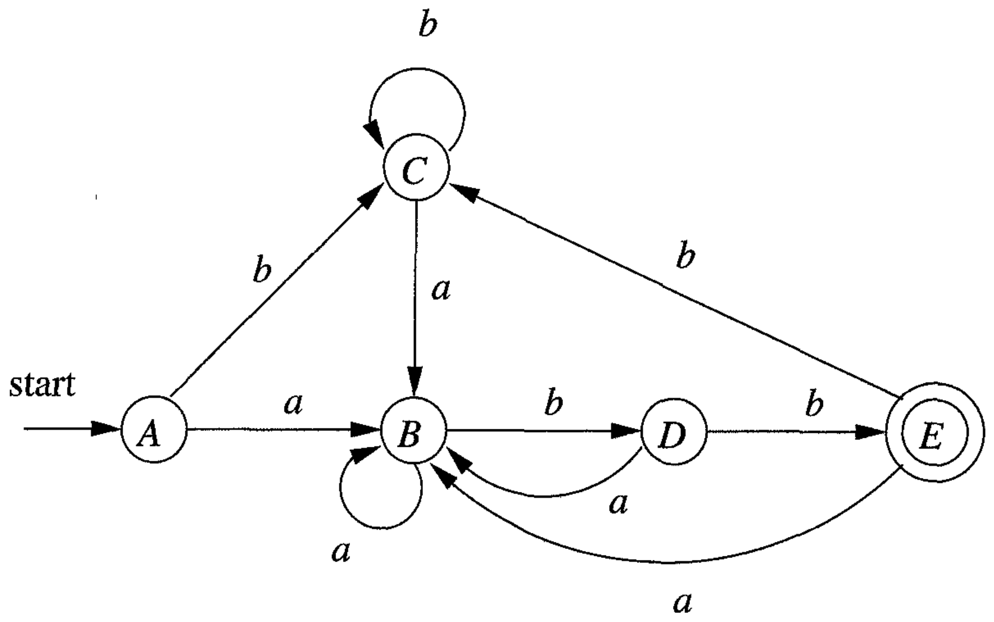
Step1: Partition graph π into 2 groups (accepting, non-accepting)

Example

- {A, B, C, D} → non-accepting state
- {E} → accepting state

Construct a new graph

- $\pi_{new} = \{ \{A, B, C, D\} \{E\} \}$



DFA Minimization

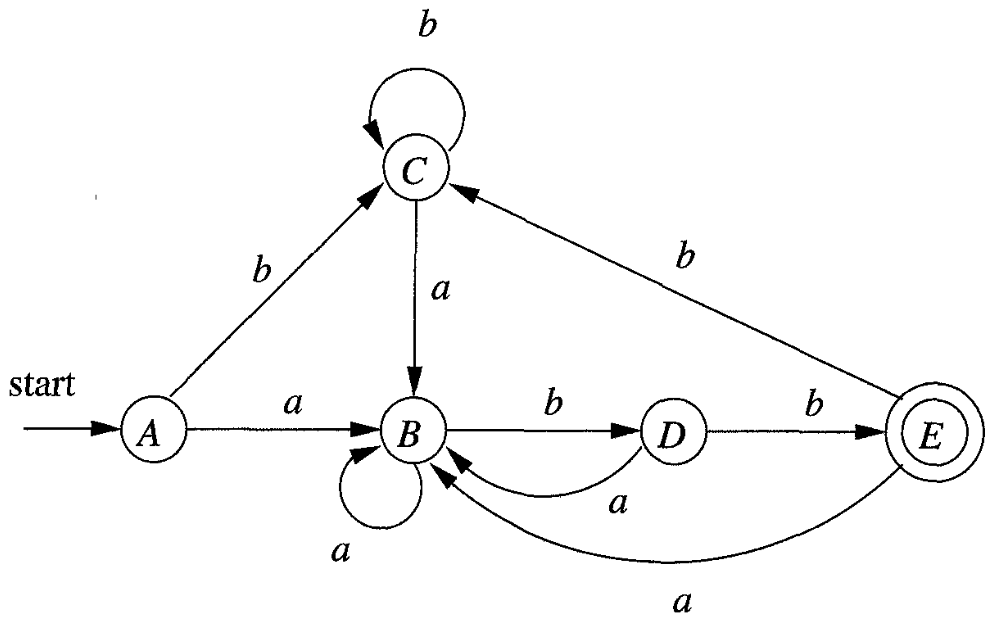
Step2: For each group G in π_{new}

- Split group G into subgroups
  - 2 states s & t in same subgroup <u>iff</u> for all input symbol a, state s & t have transitions on a to states in the same group

Example

- For {A, B, C, D}
  - On input a ➔ {A, B, C, D} go to B
  - On input b ➔ {A, B, C} go to {A, B, C, D}

    ➔ {D} go to {E}



π_{new} = { {A, B, C} {D} {E} }

DFA Minimization

Step3: Repeat Step2 till converge

Example

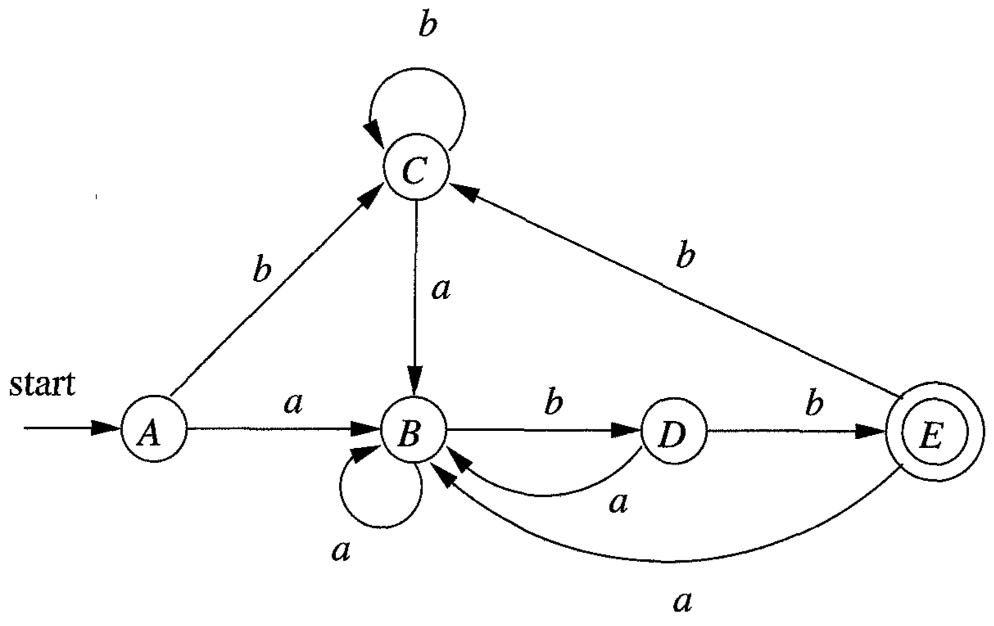
- For {A, B, C}
  - On input b ➔ {A, C} go to {A, B, C}

    ➔ {B} go to {D}

π_{new} = { {A, C} {B} {D} {E} }

- For {A, C}
  - On input a ➔ {A, C} go to {A, C}
  - On input b ➔ {A, C} go to {A, C}

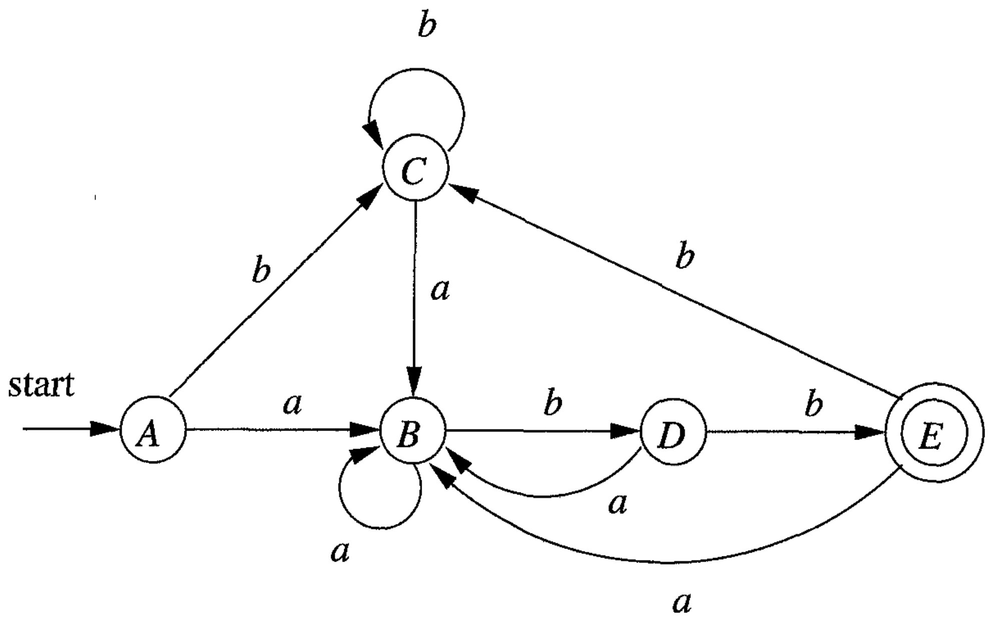
π_{new} = { {A, C} {B} {D} {E} } = π_{final}



DFA Minimization

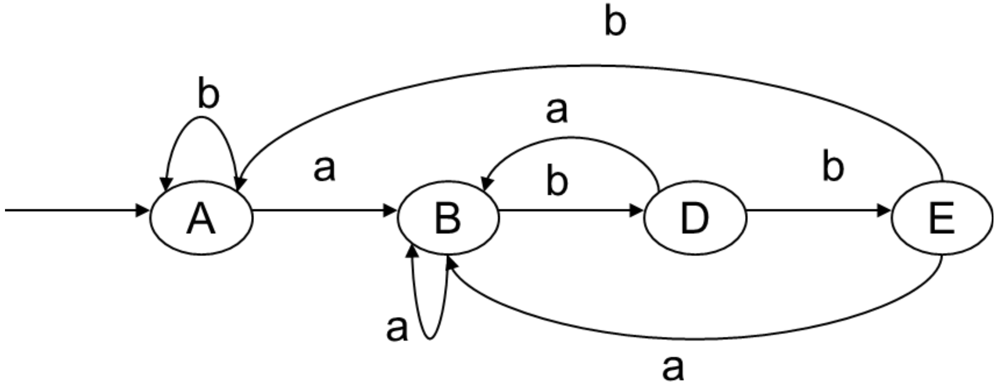
Step4: Choose 1 state in each group in π_{final}

- Start State
- Accepting State
- Transition from that state to another outside on input a

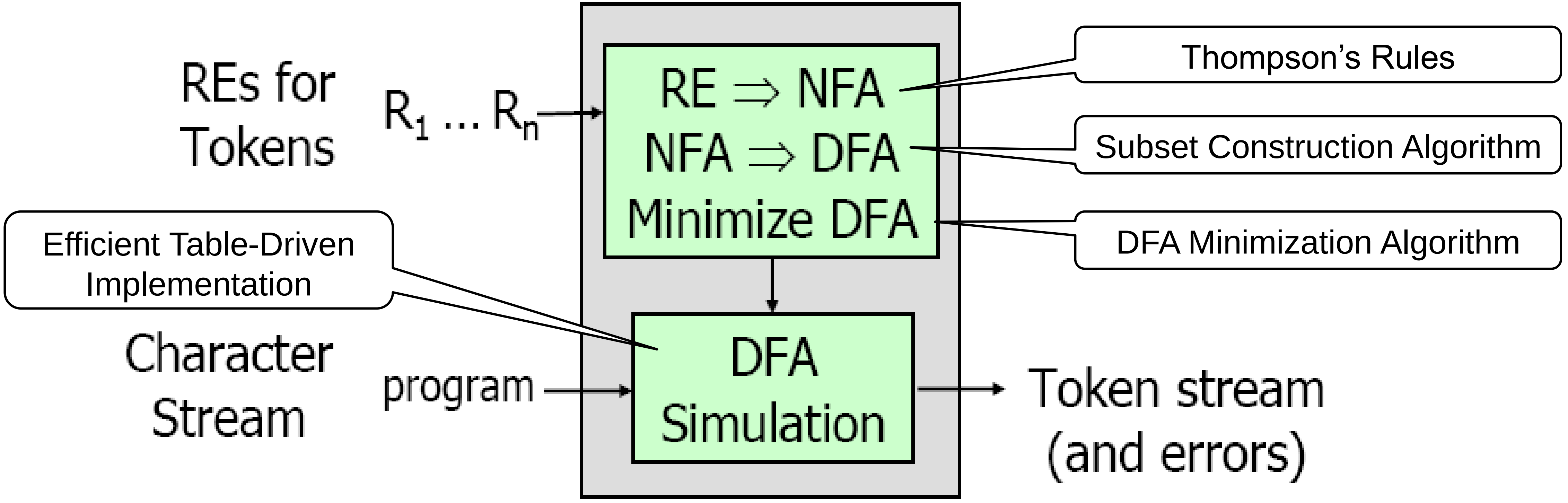


Example

- π_{final} = { {A, C} {B} {D} {E} }
- Pick A



Summary


REs for Tokens
$R_1 \dots R_n$
RE ⇒ NFA
NFA ⇒ DFA
Minimize DFA
Thompson's Rules
Subset Construction Algorithm
DFA Minimization Algorithm
Efficient Table-Driven Implementation
Character Stream
program
DFA Simulation
Token stream (and errors)

Thank you