CS 314 Principles of Programming Languages

Lecture 3: Syntax Analysis (Scanning)

Prof. Zheng Zhang

Rutgers University

September 12, 2018

Class Information

Homework 1

- Due 9/18 11:55pm EST.
- Only accepted in **pdf** format.
- No late submission will be accepted.

TA office hours announced

- See Sakai course page

Review: Formalisms for Lexical and Syntactic Analysis

Two issues in ***Formal Languages:***

- *Language Specification* → formalism to describe what a valid program (word/sentence) looks like.

- *Language Recognition* → formalism to describe a machine and an algorithm that can verify that a program is valid or not.

We use **regular expression** to specify tokens (words)

A Formal Definition

Regular Expressions (RE) over an Alphabet Σ

If $\underline{x} = \mathrm{a} \in \Sigma$, then $\underline{x}$ is an **RE** denoting the set { a }
or, the language L = { a }

Assuming $\underline{x}$ and $\underline{y}$ are both **RE**s then

1. $\underline{xy}$ is an **RE** denoting $L(\underline{x})L(\underline{y}) = \{\, pq \mid p \in L(\underline{x}) \text{ and } q \in L(\underline{y}) \,\}$
2. $\underline{x} \mid \underline{y}$ is an **RE** denoting $L(\underline{x}) \cup L(\underline{y})$

3. $\underline{x}^*$ is an **RE** denoting

$$L(\underline{x})^* = \cup_{\mathbf{0} \le k < \infty} L(\underline{x})^k$$ *(Kleene Closure)*

Set of all strings that are zero or more concatenations of $\underline{x}$

4. $\underline{x}^+$ is an **RE** denoting

$$L(\underline{x})^+ = \cup_{\mathbf{1} \le k < \infty} L(\underline{x})^k$$ *(Positive Closure)*

Set of all strings that are one or more concatenations of $\underline{x}$

ε is an **RE** denoting the empty set

Review: Regular Expressions

A syntax (notation) to specify regular languages.

RE p	**Language L(*p*)**
$\underline{x} \mid \underline{y}$	$L(\underline{x}) \cup L(\underline{y})$
$\underline{xy}$	$\{RS \mid R \in L(\underline{x}), S \in L(\underline{y})\}$
$\underline{x}+$	$L(\underline{x}) \cup L(\underline{xx}) \cup L(\underline{xxx}) \cup \ldots$
$\underline{x}^* \; (\underline{x}^* = \underline{x}^+ \mid \epsilon)$	$\{\epsilon\} \cup L(\underline{x}) \cup L(\underline{xx}) \cup \ldots$
(s)	$L(s)$
$\mathbf{a}$	$\{\mathbf{a}\}$
ϵ	$\{\epsilon\}$

The symbols underlined denotes a regular expression, i.e., $\underline{x}$

*The symbols in bold-face denotes a letter from the alphabet, i.e., **a***

Review: Formalisms for Lexical and Syntactic Analysis

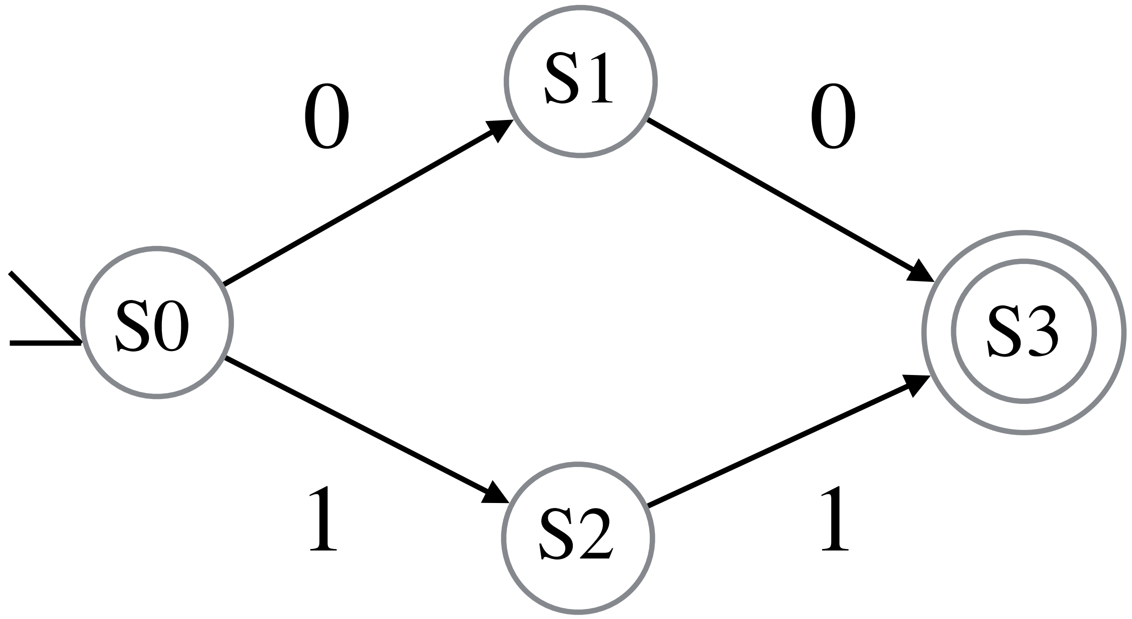
Two issues in ***Formal Languages:***

- *Language Specification* → formalism to describe what a valid program (word/sentence) looks like.

- *Language Recognition* → formalism to describe a machine and an algorithm that can verify that a program is valid or not.

We use **finite state automata** to recognize regular language

Finite State Automata



A Finite-State Automaton is a quadruple: < S, s, F, T >

- S is a set of states, e.g., {S0, S1, S2, S3}
- s is the start state, e.g., S0
- F is a set of final states, e.g., {S3}
- T is a set of labeled transitions, of the form (state, input) → state [i.e., $S \times \Sigma \rightarrow S$]

Recognizers for Regular Expressions

Let *letter* stand for A | B | C | . . . | Z

Let *digit* stand for 0 | 1 | 2 | . . . | 9

Integer Constant

Regular Expression: digit^{+}

FSA:

From RE to Scanner

Classic approach is a three-step method:

1. Build automata for each piece of the **RE** using a **template**.
   Multiple automata can be pasted using ε-transition.
   This construction is called "**Thompson's construction**"
2. Convert the newly built automaton into a deterministic automaton.
   This construction is called the "**subset construction**"
3. Given the deterministic automaton, minimize the number of states.
   Minimization is a **space optimization**.

Non-deterministic Finite Automaton (NFA)

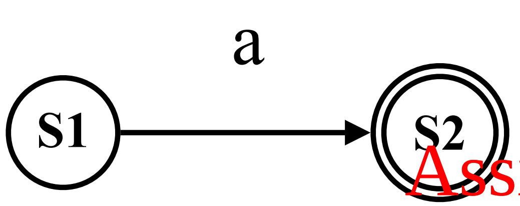
- NFA might have transitions on ε
- Non-deterministic choice: multiple transition from the same sate on the same symbol

Deterministic finite automaton (DFA) has no ε-transitions and all choices are single-valued.

Thompson's Construction

- From each RE symbol and operator, we have a template
- Build them, in precedence order, and join them with ε-transition

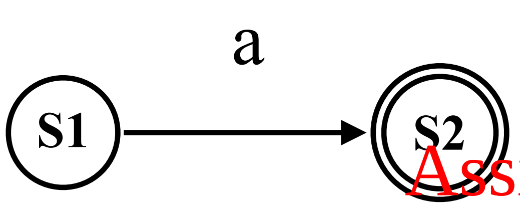
NFA for **a**



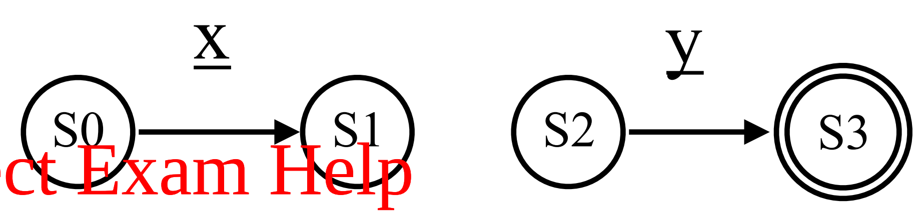
Thompson's Construction

- From each RE symbol and operator, we have a template
- Build them, in precedence order, and join them with ε-transition

NFA for **a**



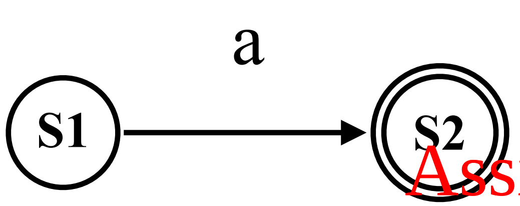
NFA for xy



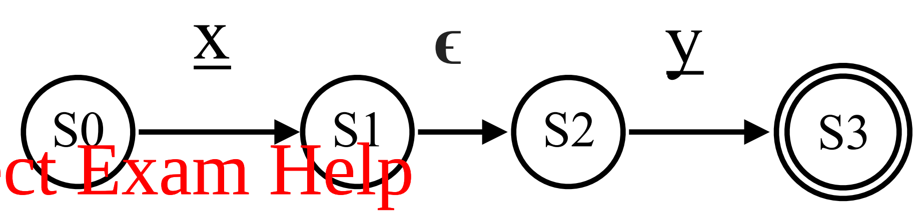
Thompson's Construction

- From each RE symbol and operator, we have a template
- Build them, in precedence order, and join them with ε-transition

NFA for **a**



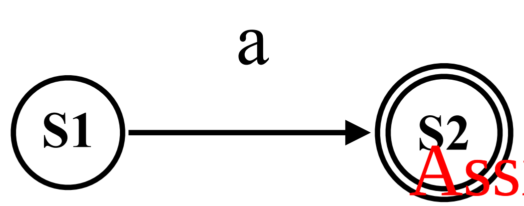
NFA for xy



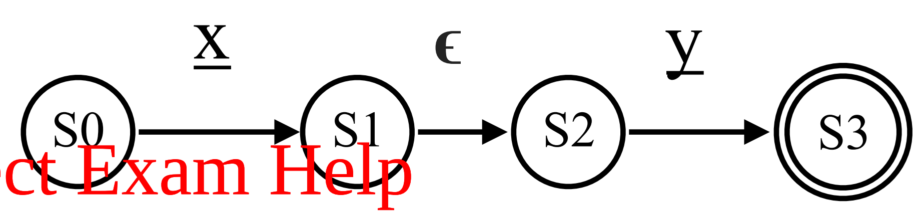
Thompson's Construction

- From each RE symbol and operator, we have a template
- Build them, in precedence order, and join them with ε-transition

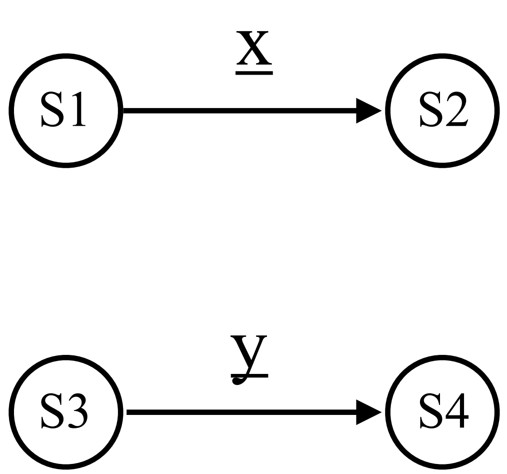
NFA for **a**



NFA for xy



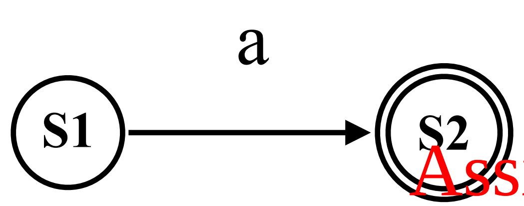
NFA for x|y



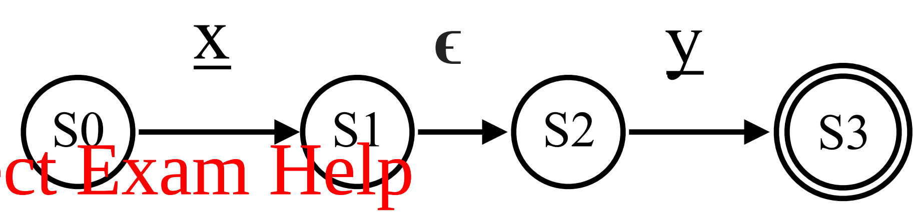
Thompson's Construction

- From each RE symbol and operator, we have a template
- Build them, in precedence order, and join them with ε-transition

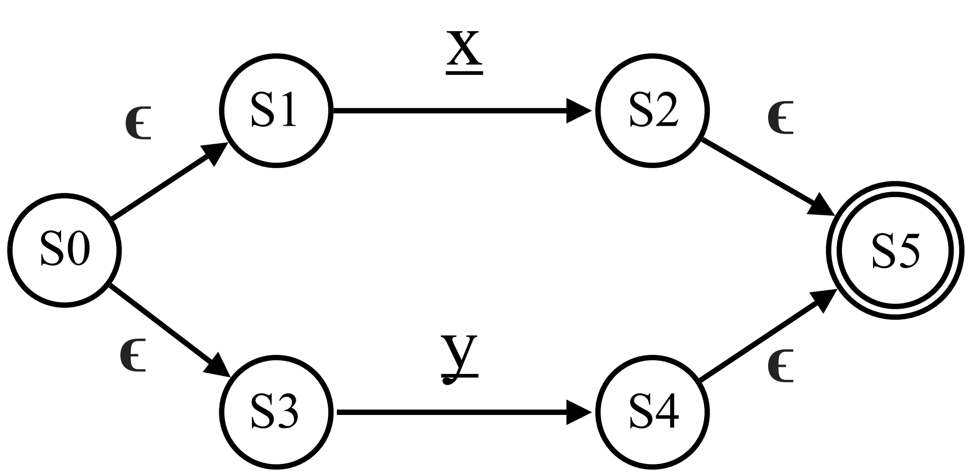
NFA for **a**



NFA for xy



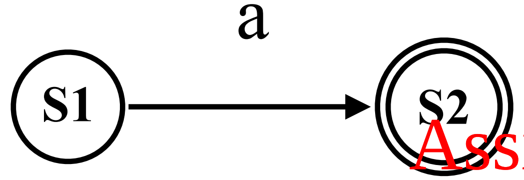
NFA for x|y



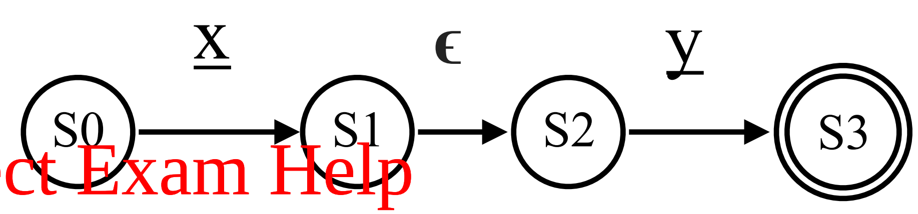
Thompson's Construction

- From each RE symbol and operator, we have a template
- Build them, in precedence order, and join them with ε-transition

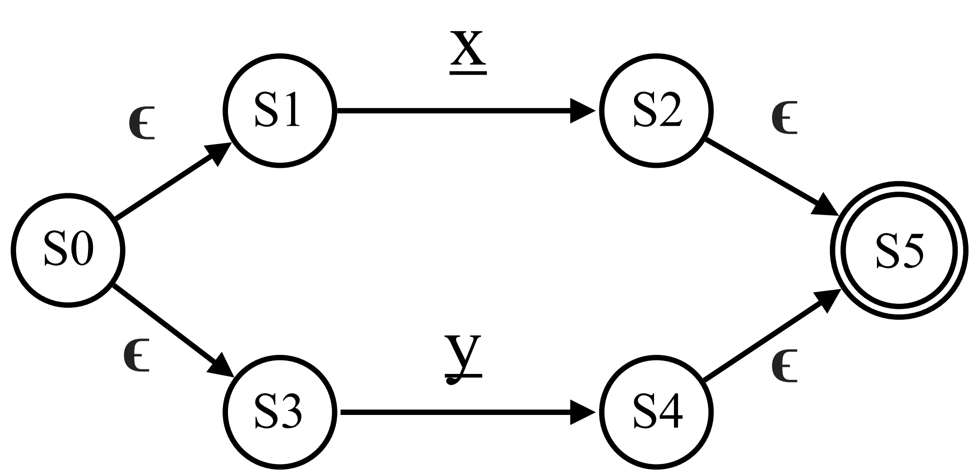
NFA for **a**



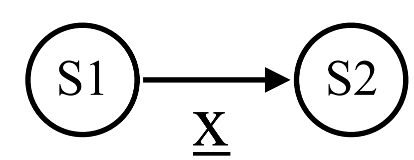
NFA for x̲y̲



NFA for x̲|y̲



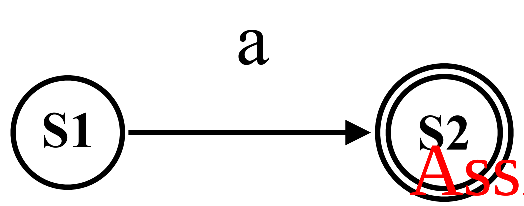
NFA for x̲*



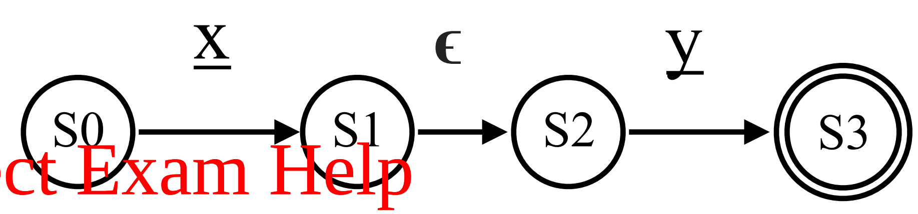
Thompson's Construction

- From each RE symbol and operator, we have a template
- Build them, in precedence order, and join them with ε-transition

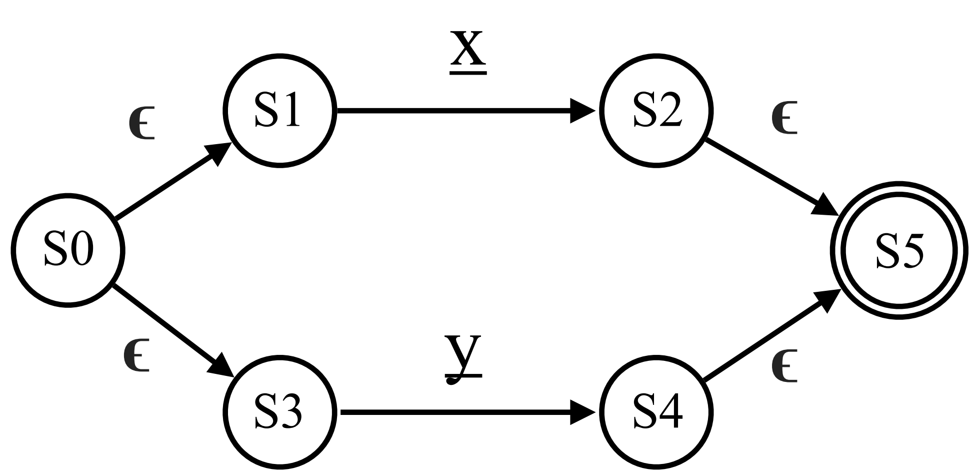
NFA for **a**



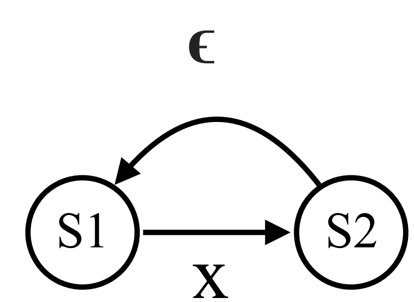
NFA for xy



NFA for x|y



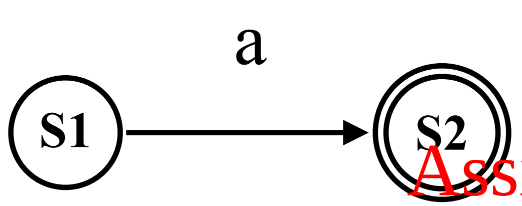
NFA for x*



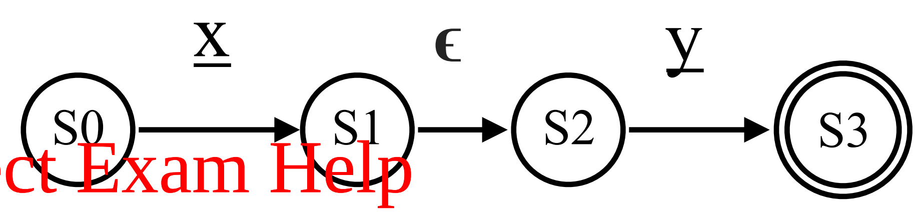
Thompson's Construction

- From each RE symbol and operator, we have a template
- Build them, in precedence order, and join them with ε-transition

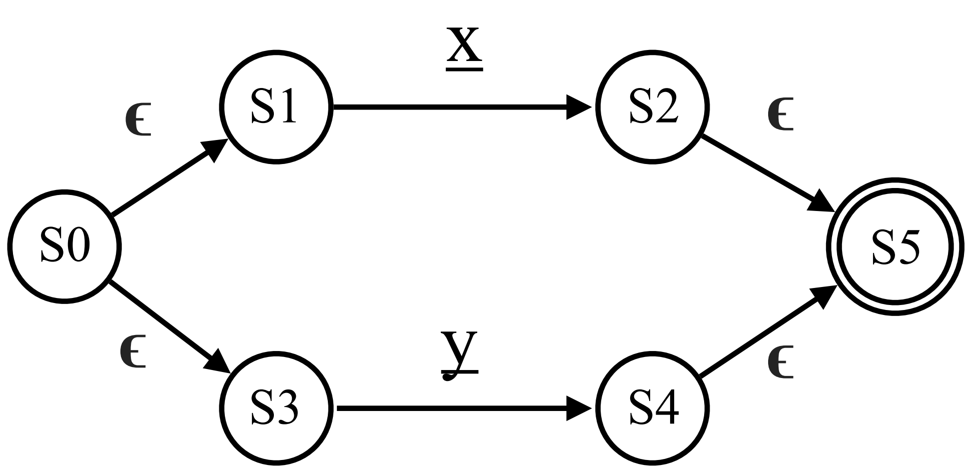
NFA for **a**



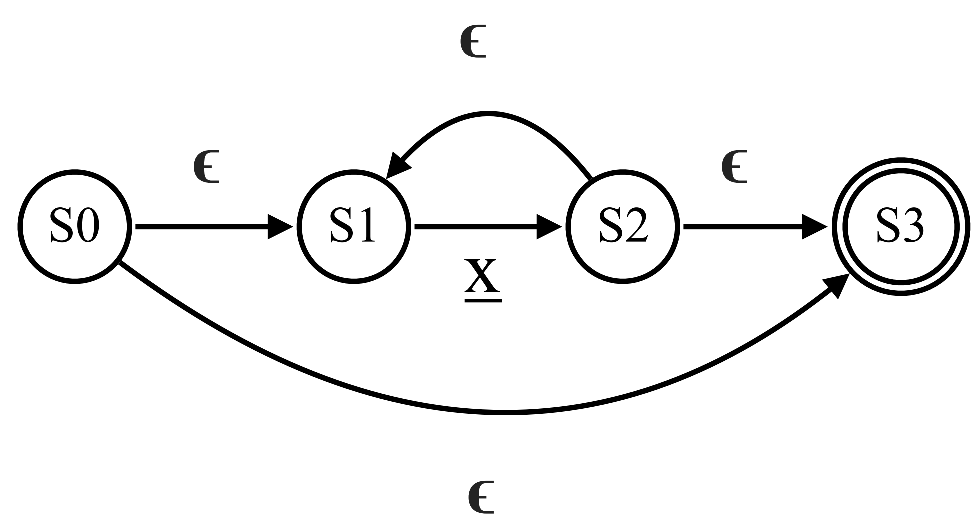
NFA for xy



NFA for x|y



NFA for x*



Thompson's Construction

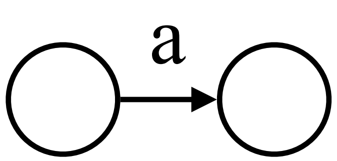
- Let's build an NFA for a (b|c)*

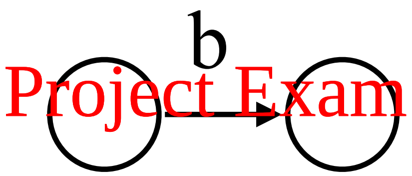
1. **a, b, & c**
2. **b | c**
3. **(b| c)***
4. **a (b| c)***

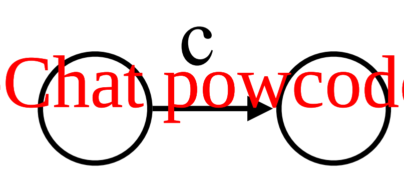
Thompson's Construction

- Let's build an NFA for a (b|c)*

1. **a, b, & c**
2. **b | c**
3. **(b| c)***
4. **a (b| c)***



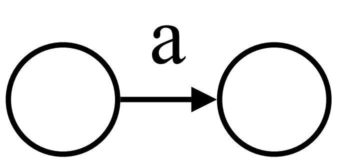


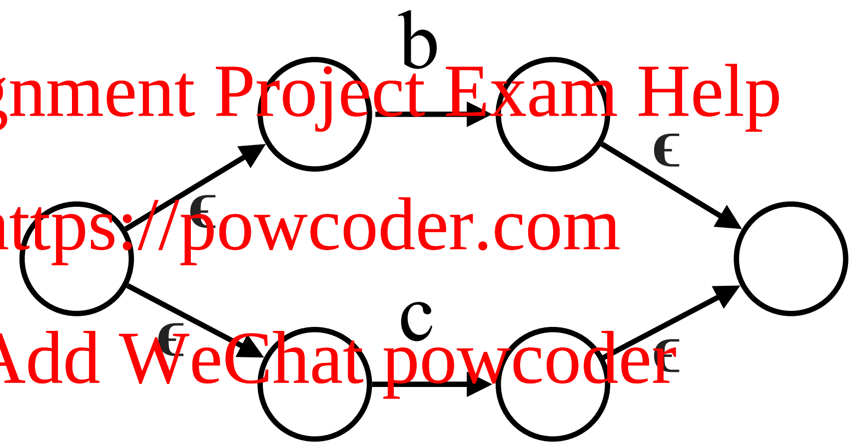


Thompson’s Construction

- Let’s build an NFA for a (b|c)*

1. **a, b, & c**
2. **b | c**
3. **(b| c)***
4. **a (b| c)***

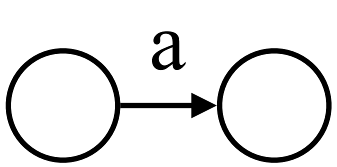


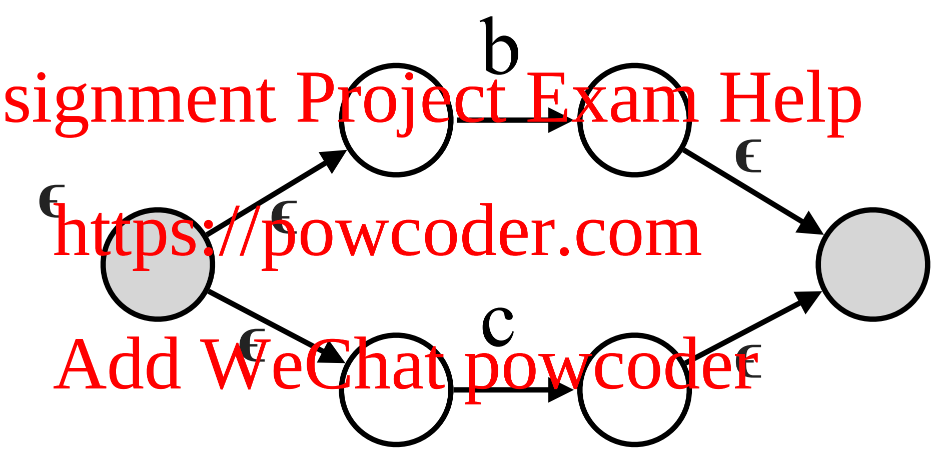


Thompson's Construction

- Let's build an NFA for a (b|c)*

1. **a, b, & c**
2. **b | c**
3. **(b| c)***
4. **a (b| c)***

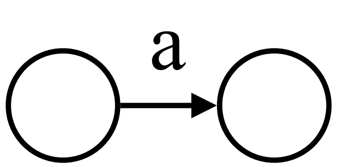


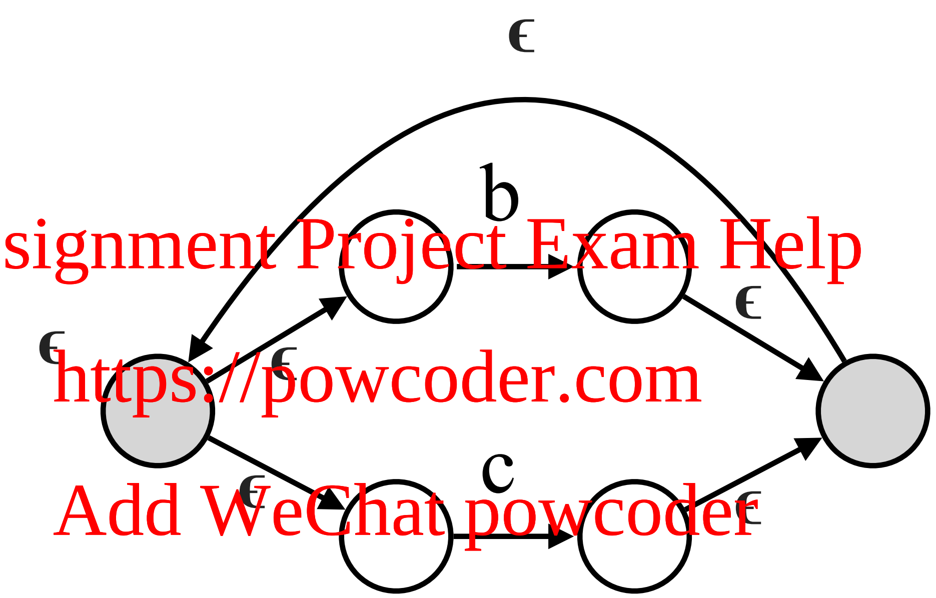


Thompson's Construction

- Let's build an NFA for a (b|c)*

1. a, b, & c
2. b | c
3. (b| c)*
4. a (b| c)*

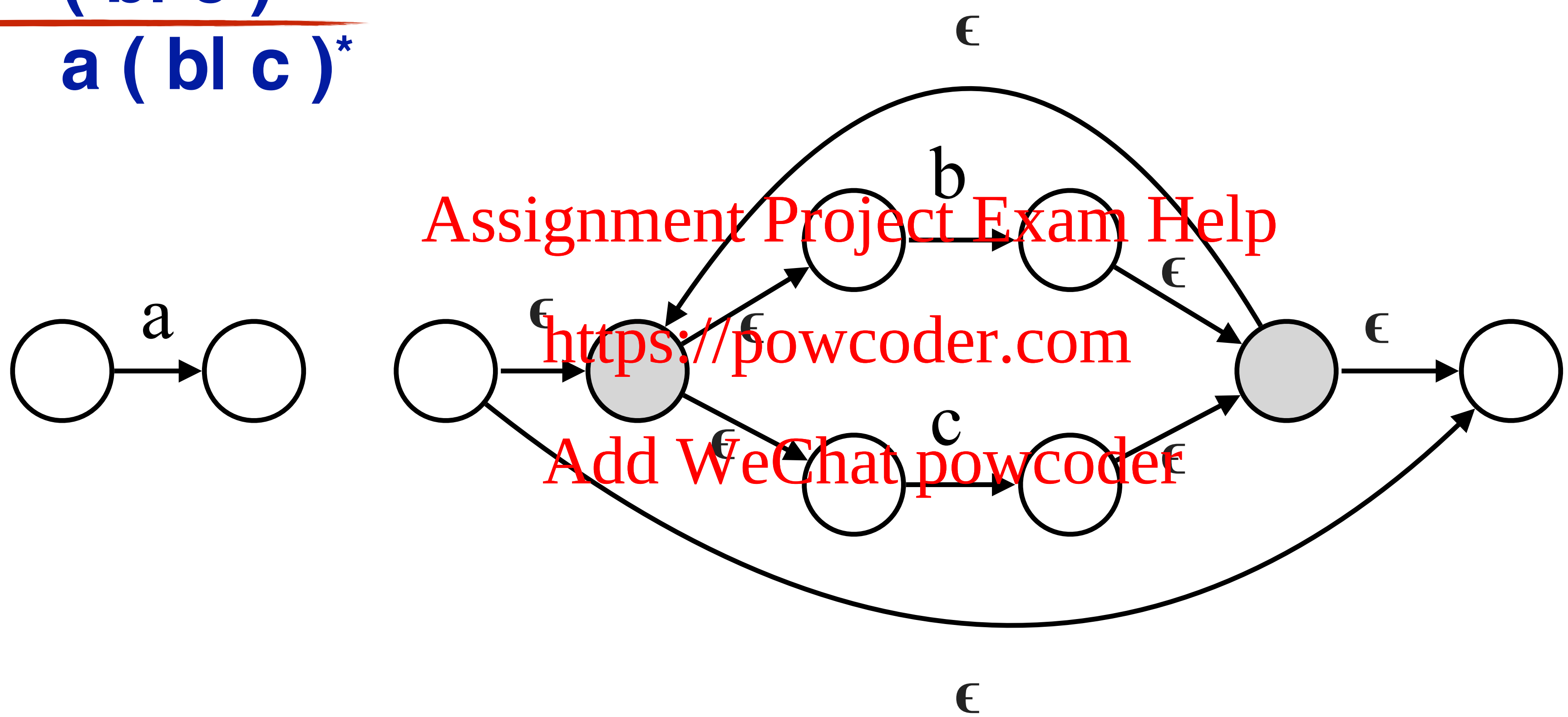




Thompson’s Construction

- Let’s build an NFA for a (b|c)*

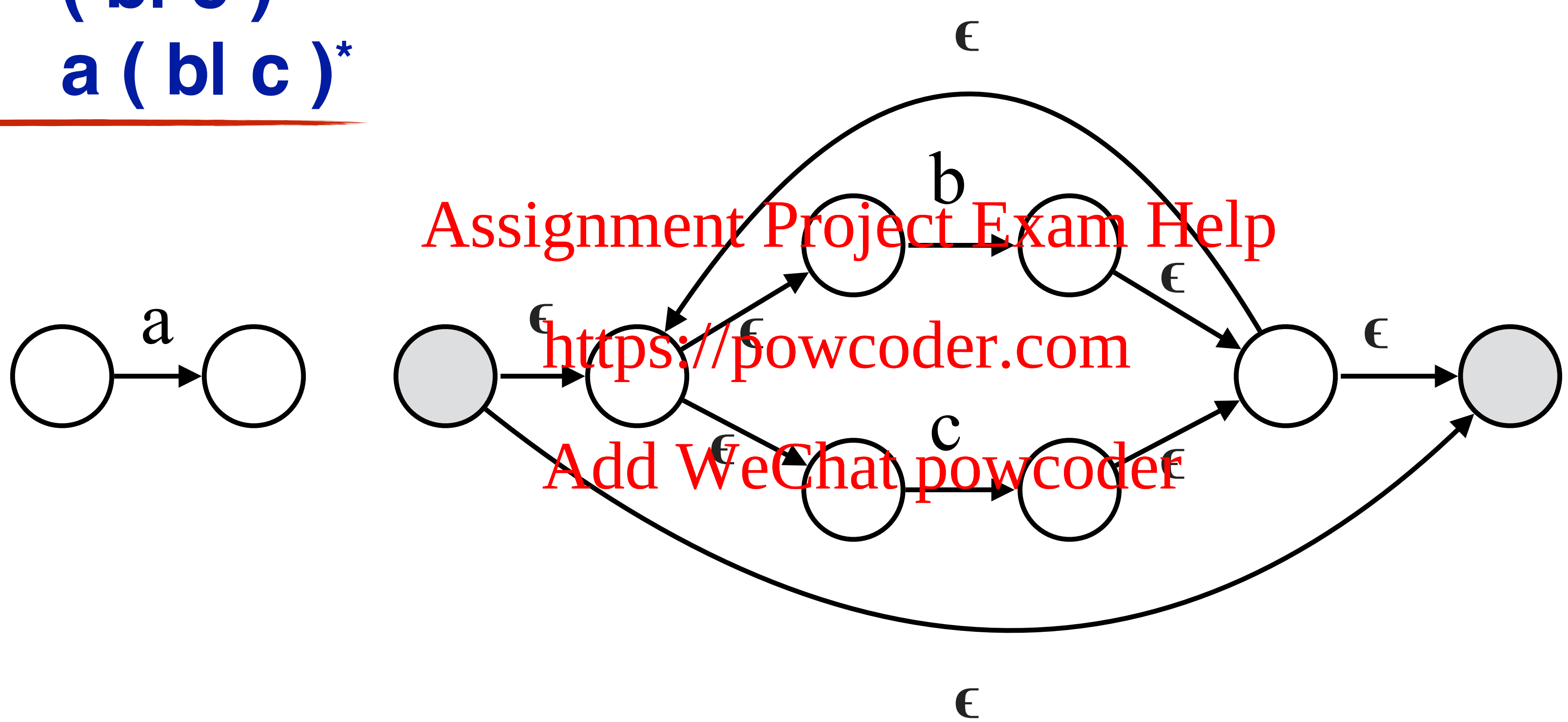
1. **a, b, & c**
2. **b | c**
3. **(b| c)***
4. **a (b| c)***



Thompson's Construction

- Let's build an NFA for a (b|c)*

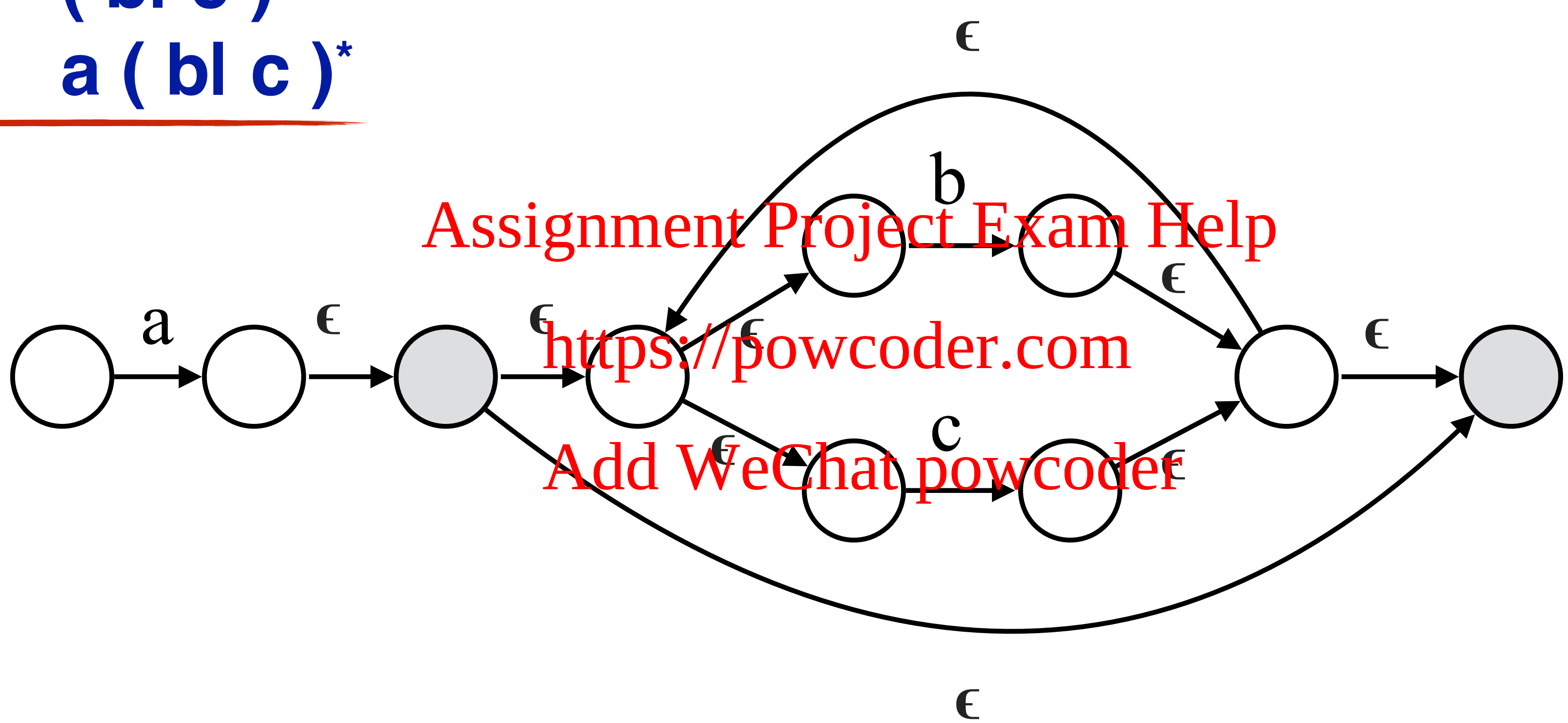
1. **a, b, & c**
2. **b | c**
3. **(b| c)***
4. **a (b| c)***



Thompson's Construction

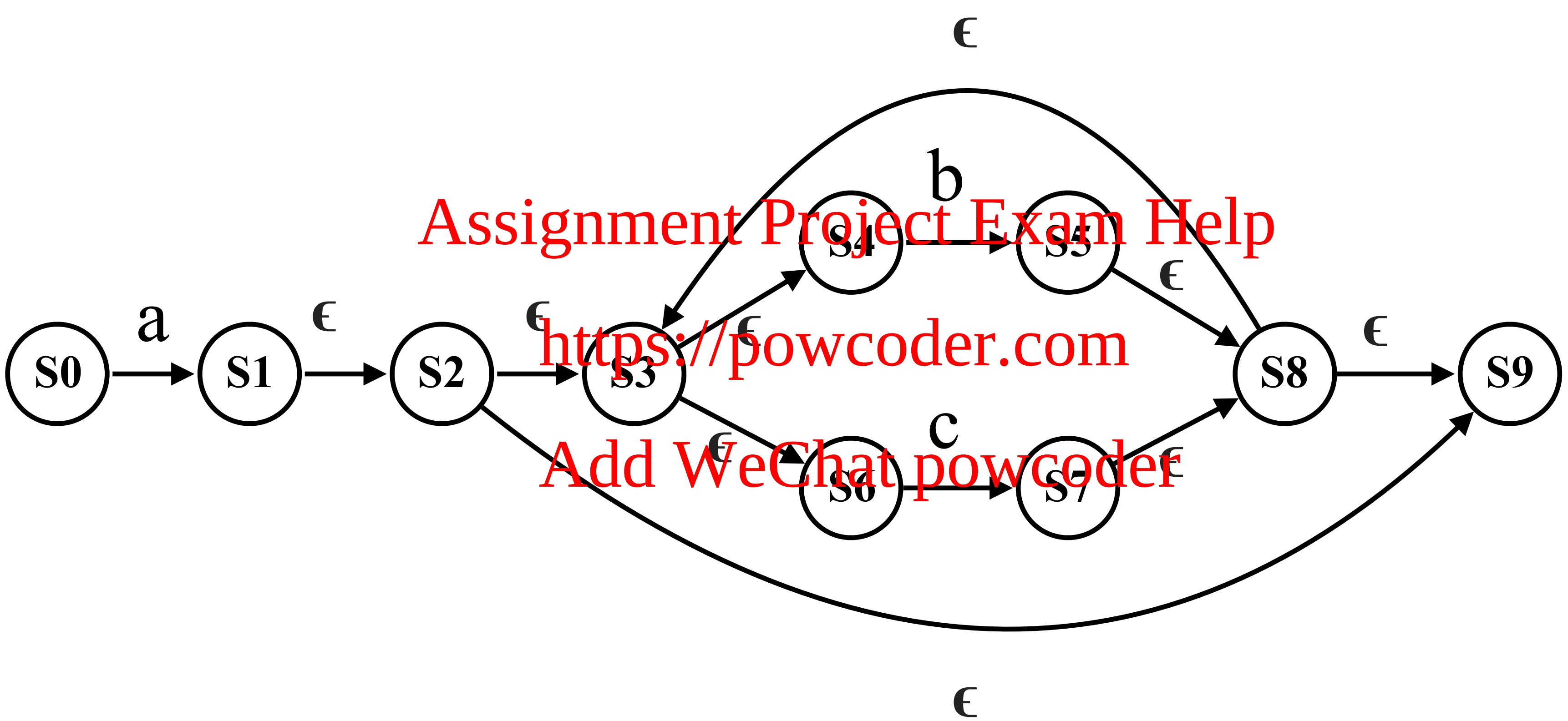
- Let's build an NFA for a (b|c)*

1. **a, b, & c**
2. **b | c**
3. **(b| c)***
4. **a (b| c)***



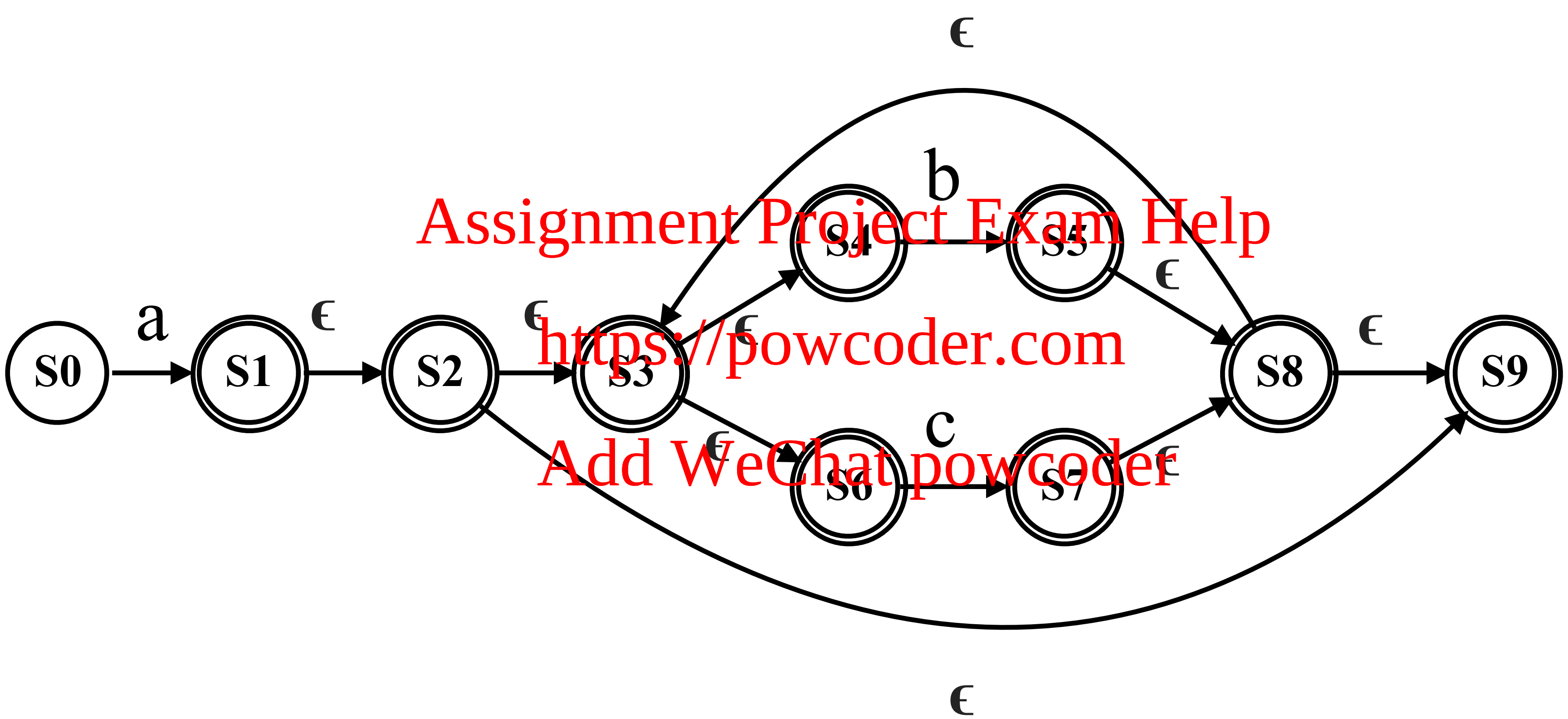
Thompson’s Construction

- Let’s build an NFA for a (b|c)*



Thompson's Construction

- Let's build an NFA for a (b|c)*



Final states are double circled

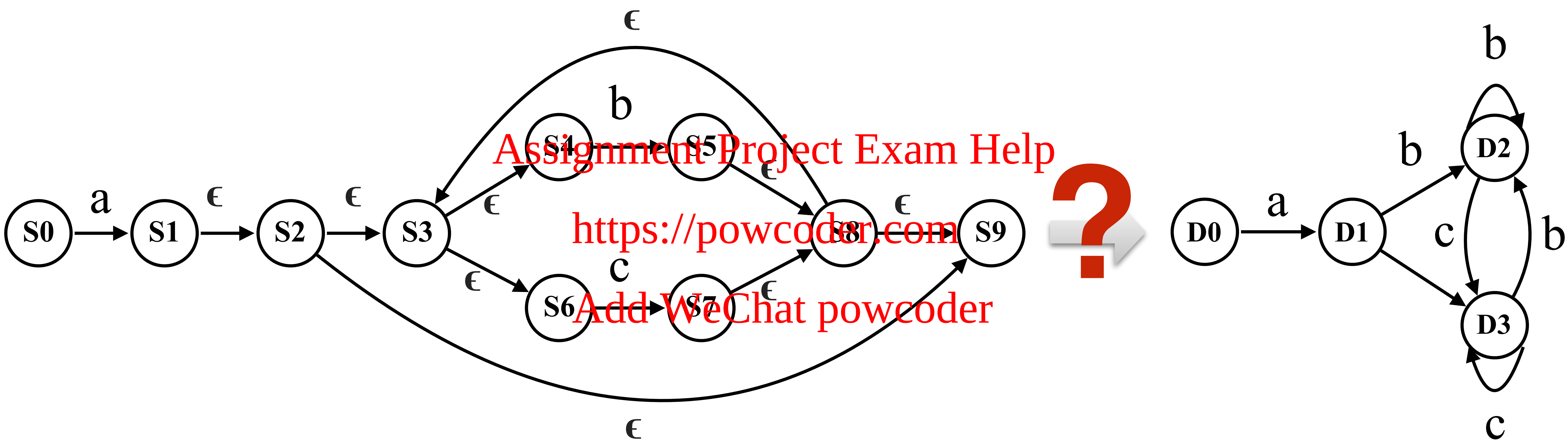
From RE to Scanner

Classic approach is a three-step method:

1. Build automata for each piece of the **RE** using a **template**.
   Multiple automata can be pasted using ε-transition.
   This construction is called "**Thompson's construction**"
2. Convert the newly built automaton into a deterministic automaton.
   This construction is called the "**subset construction**"
3. Given the deterministic automaton, minimize the number of states.
   Minimization is a **space optimization**.

Subset Construction

- Build a deterministic automaton that simulates the non-deterministic one
- Each state in the new one represents a set of states in the original one



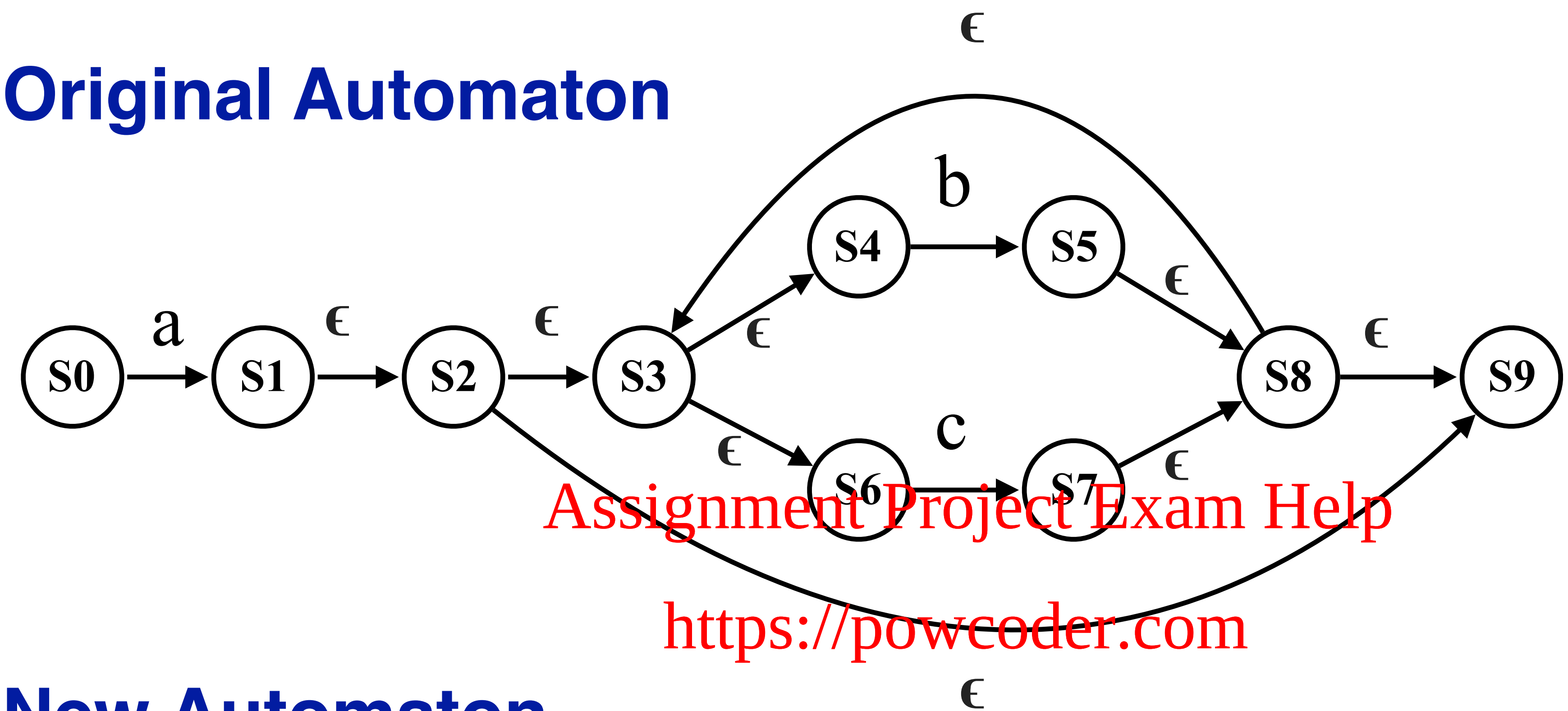
NFA

DFA

Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



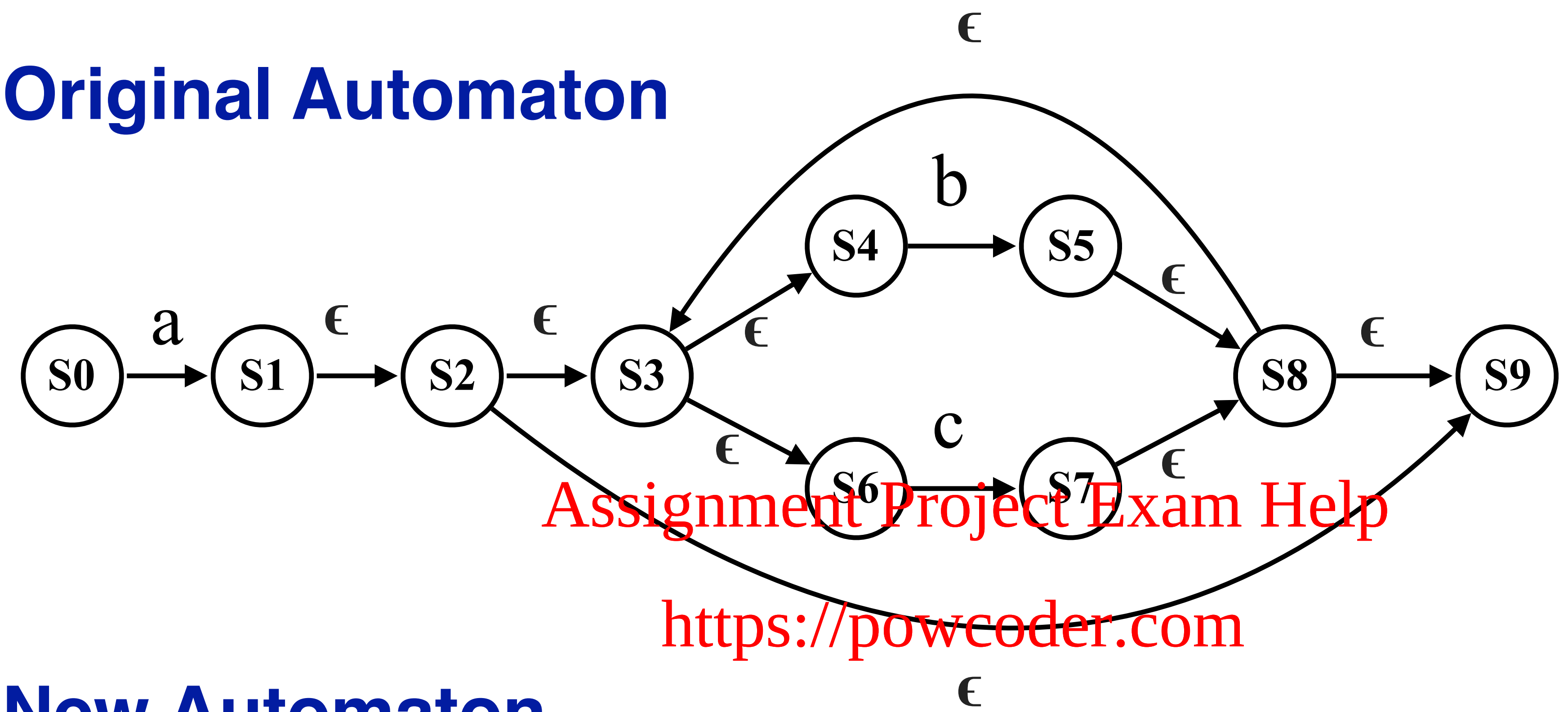
New Automaton

DFA	NFA
D0	*S0*

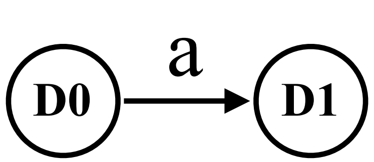
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

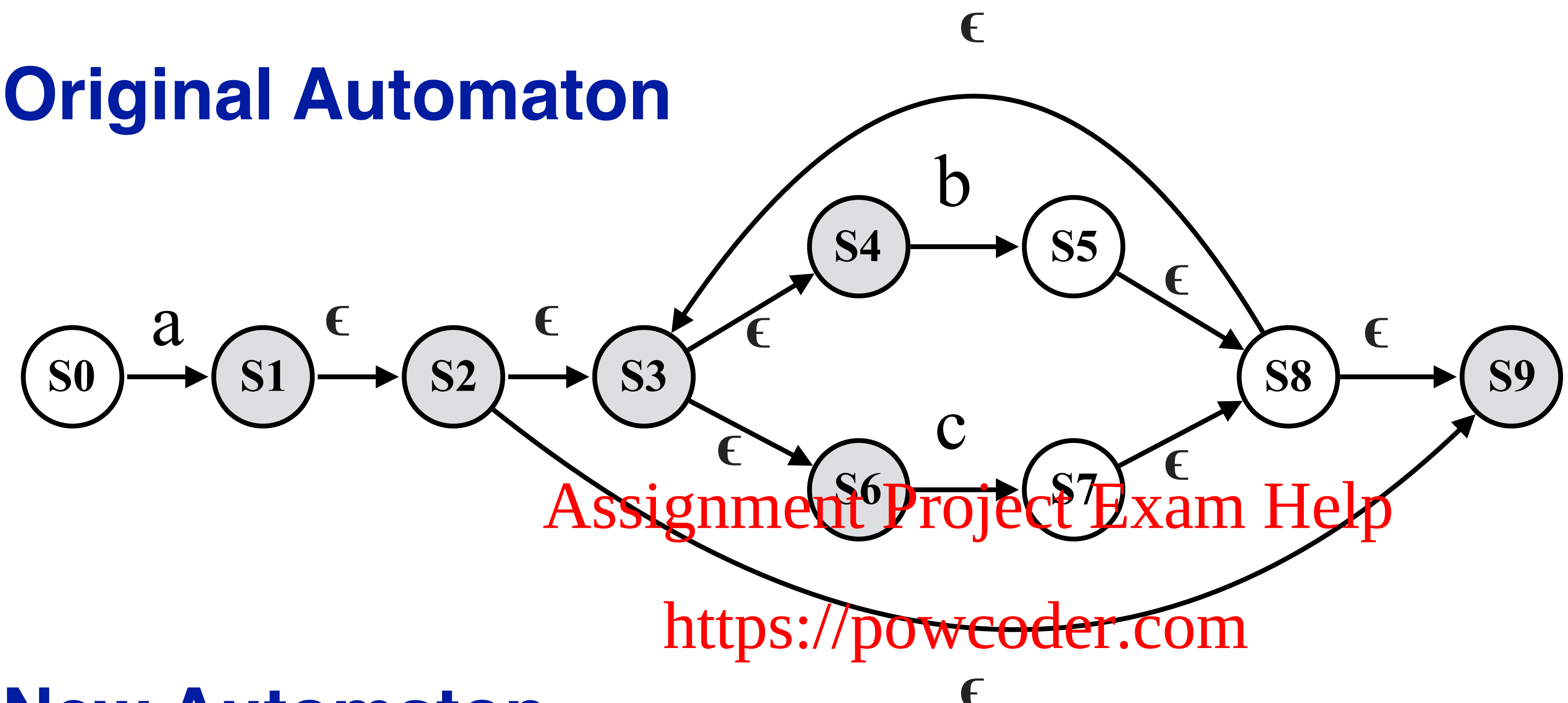


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*

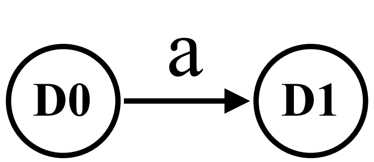
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

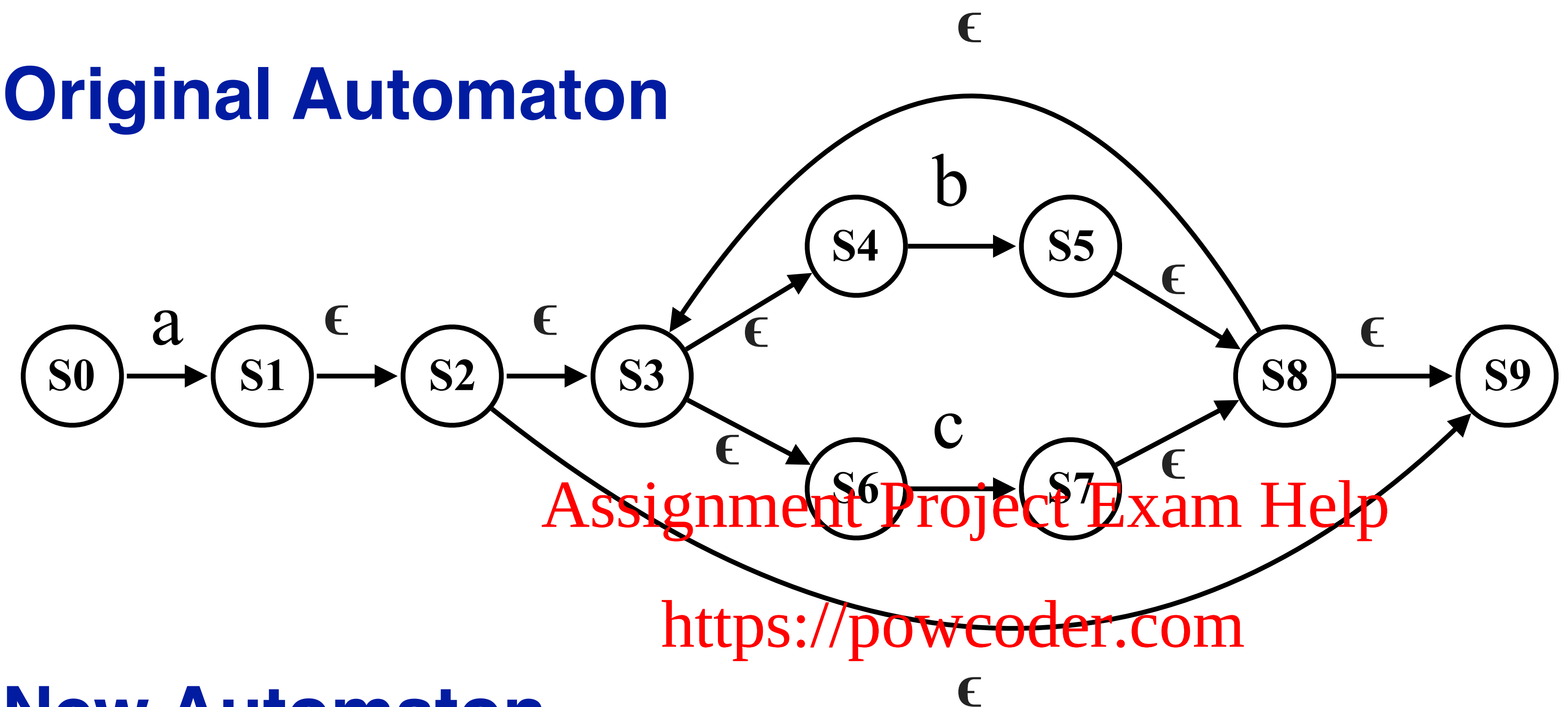


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*

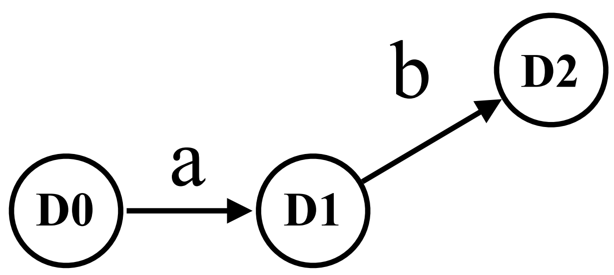
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

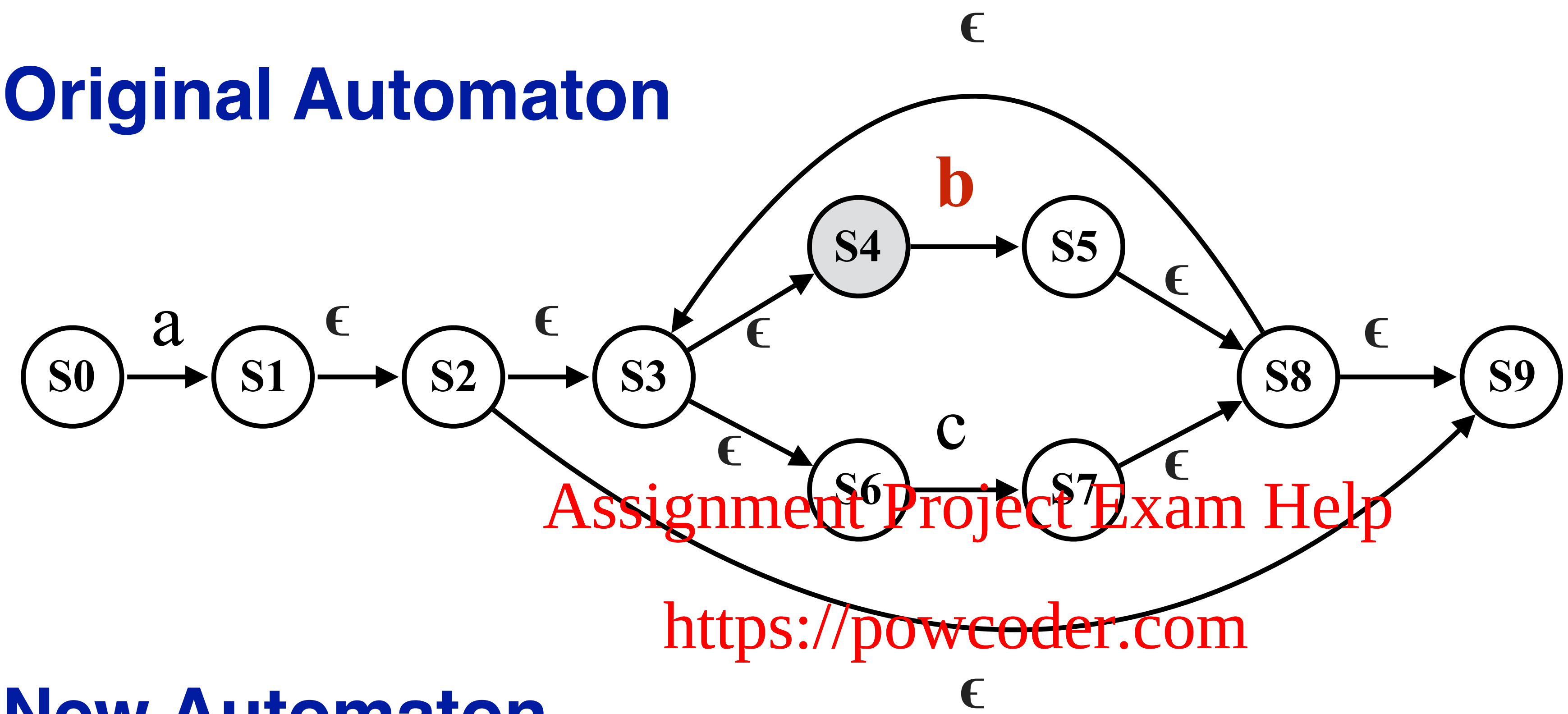


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*
D2	

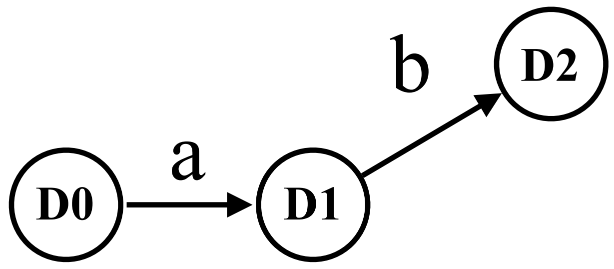
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

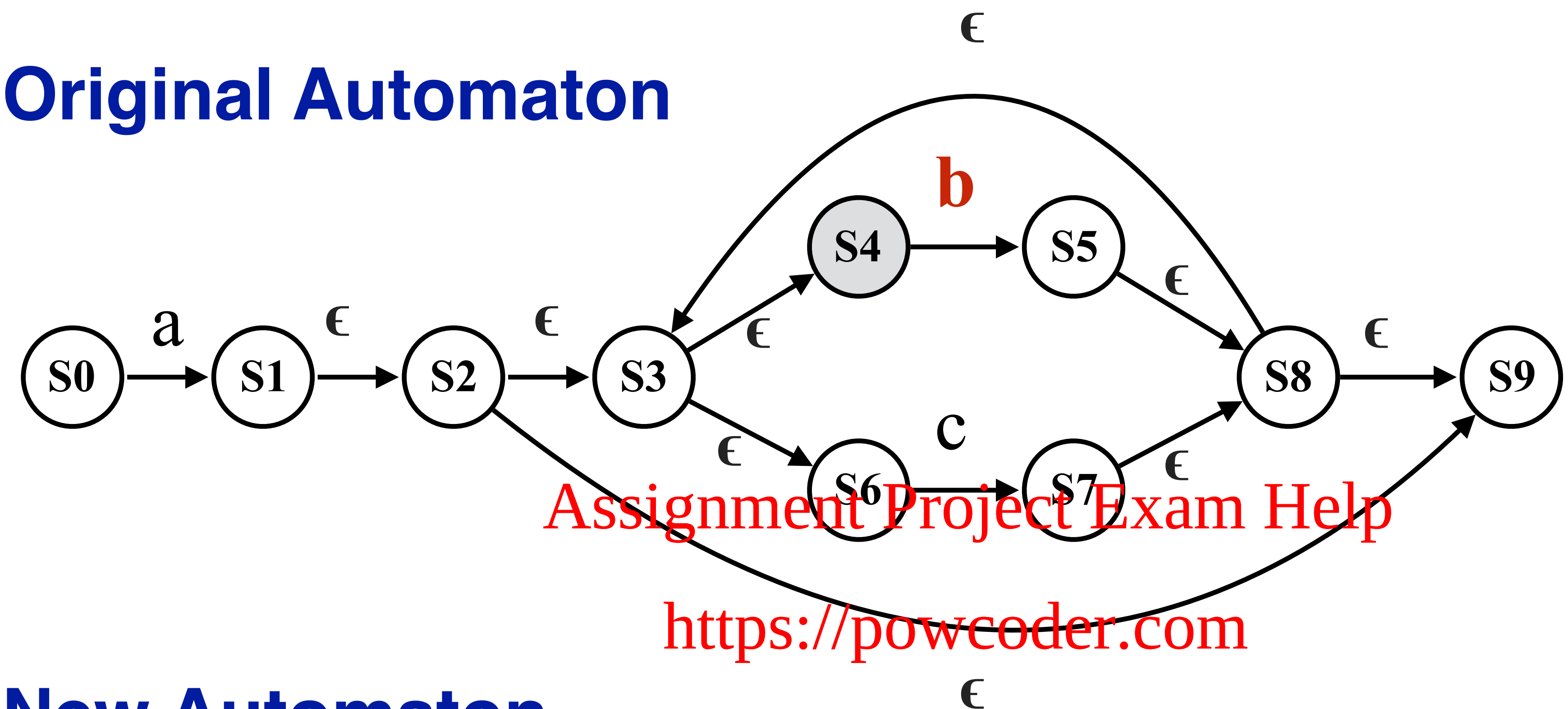


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*
D2	

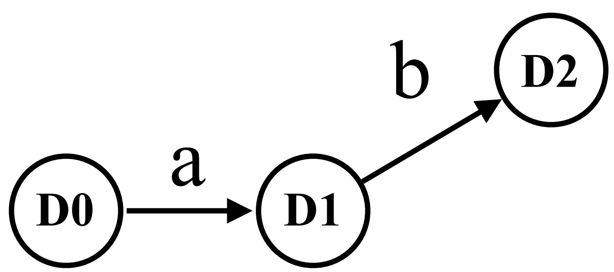
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

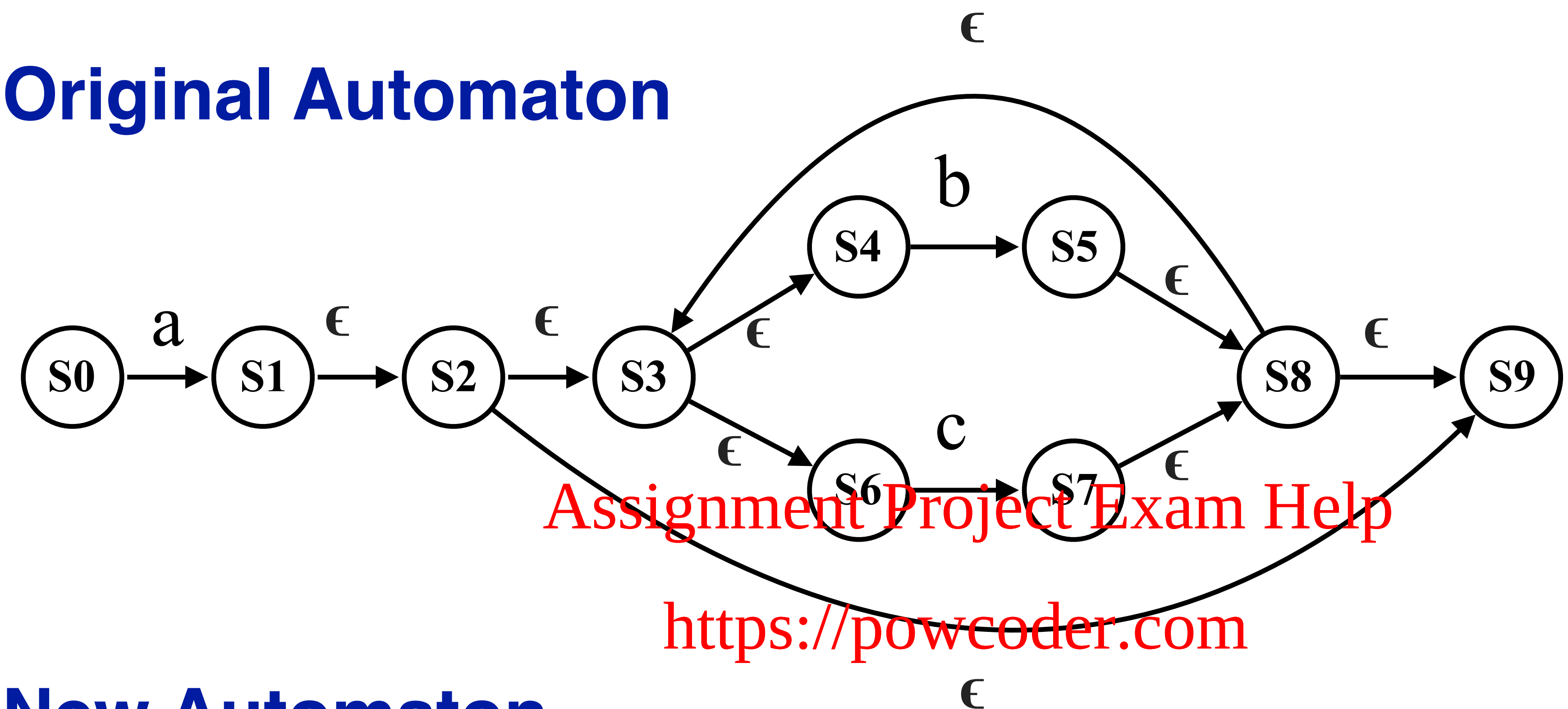


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*
D2	*S5, S8, S3, S9, S4, S6*

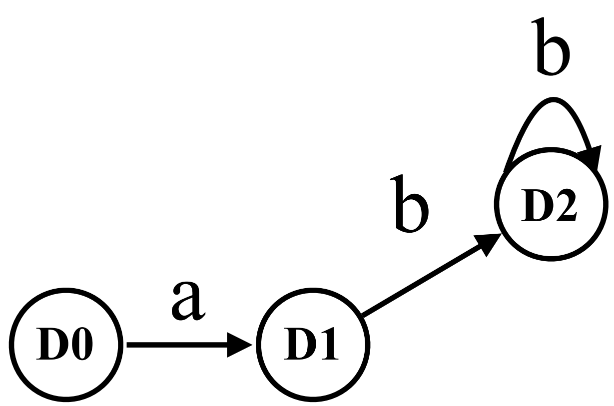
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

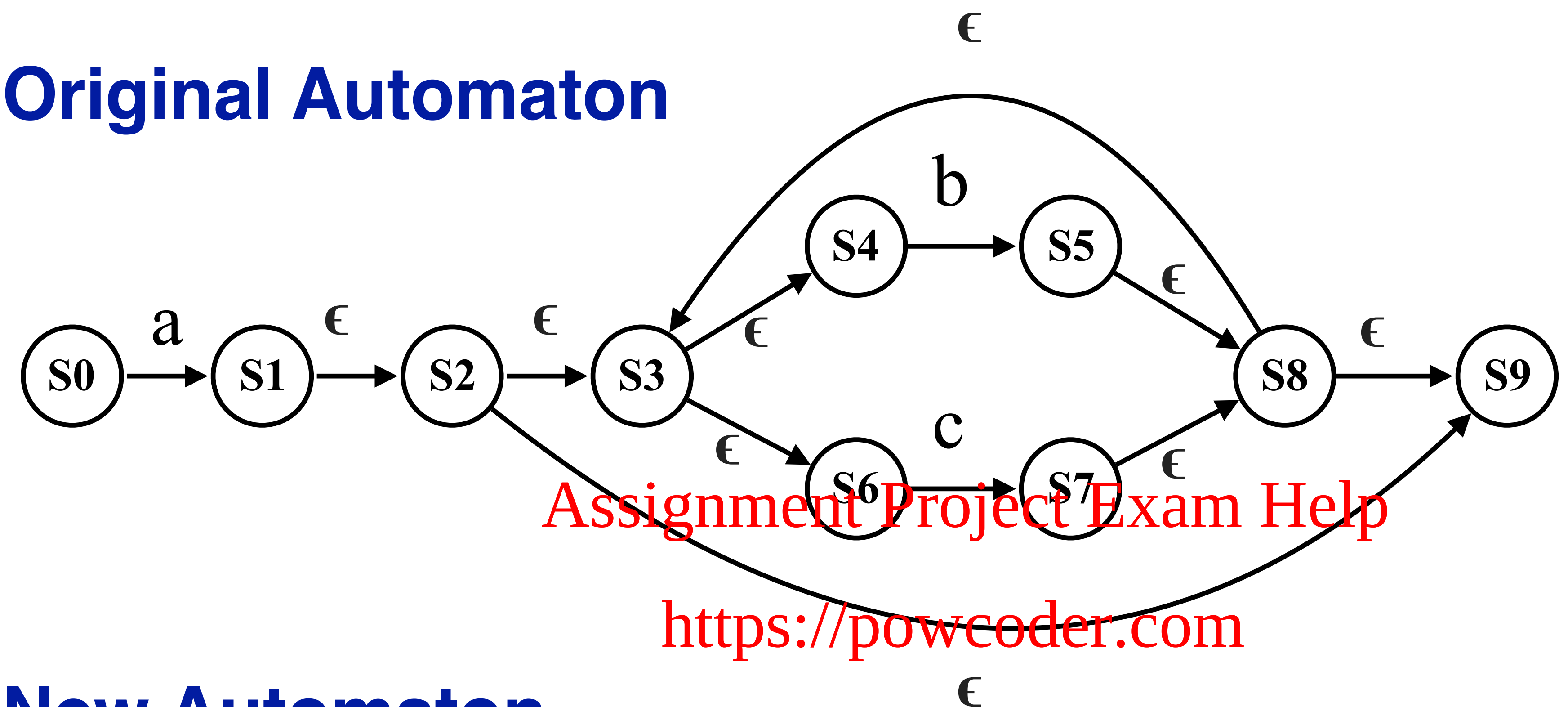


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*
D2	*S5, S8, S3, S9, S4, S6*

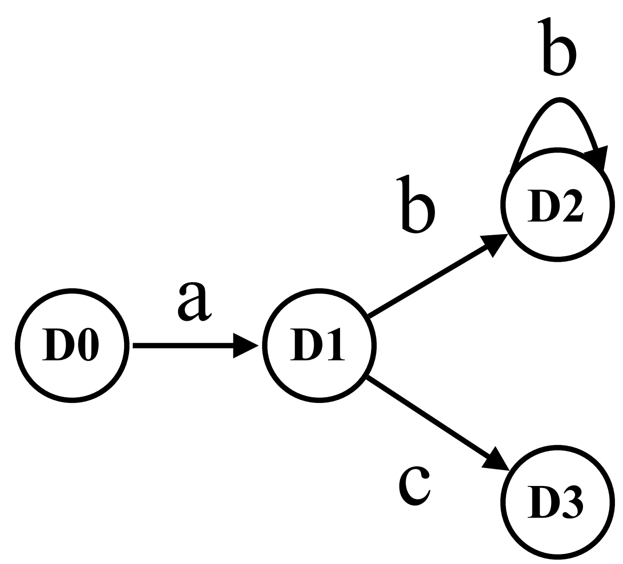
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

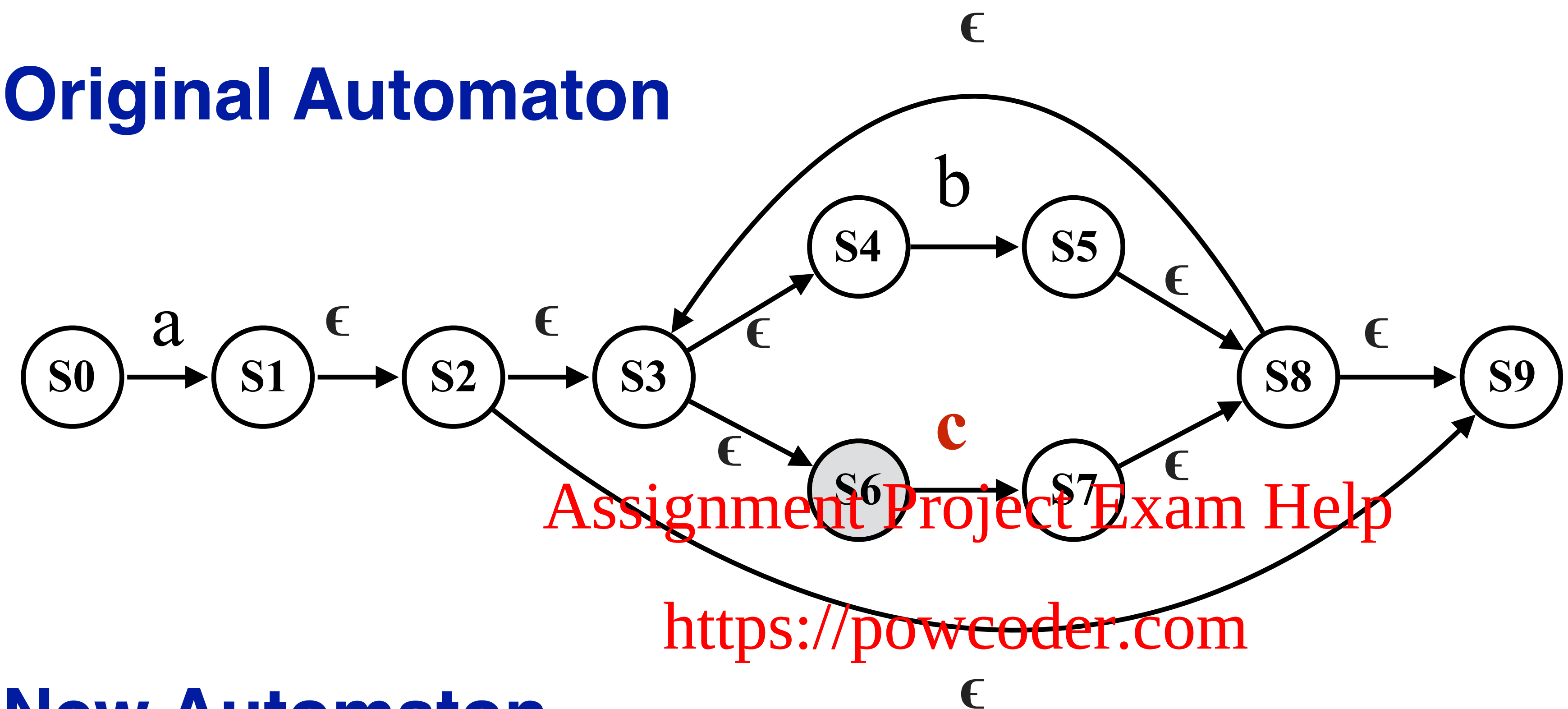


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*
D2	*S5, S8, S3, S9, S4, S6*
D3	

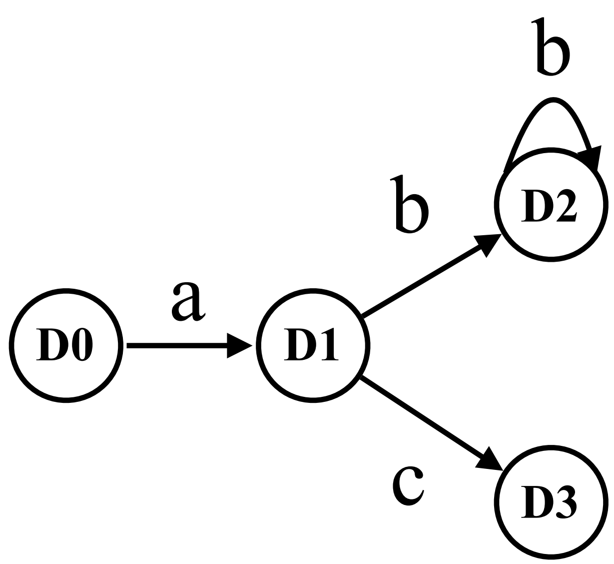
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

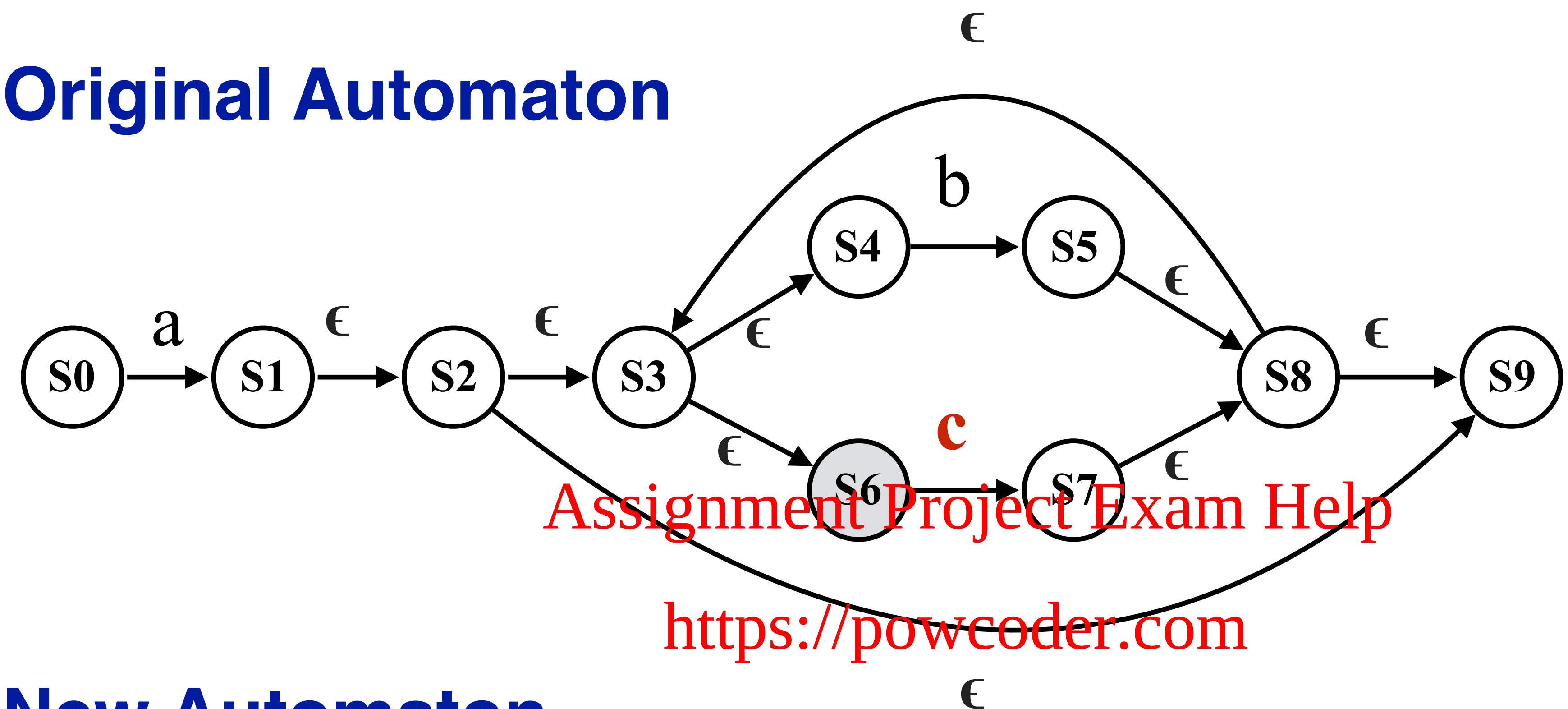


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*
D2	*S5, S8, S3, S9, S4, S6*
D3	

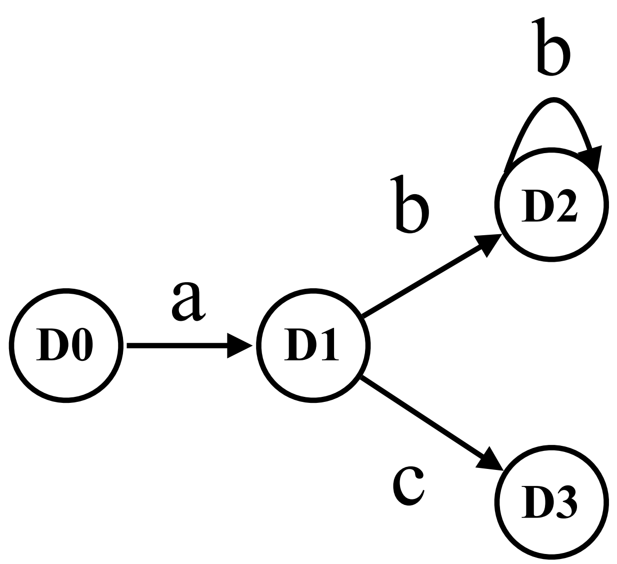
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

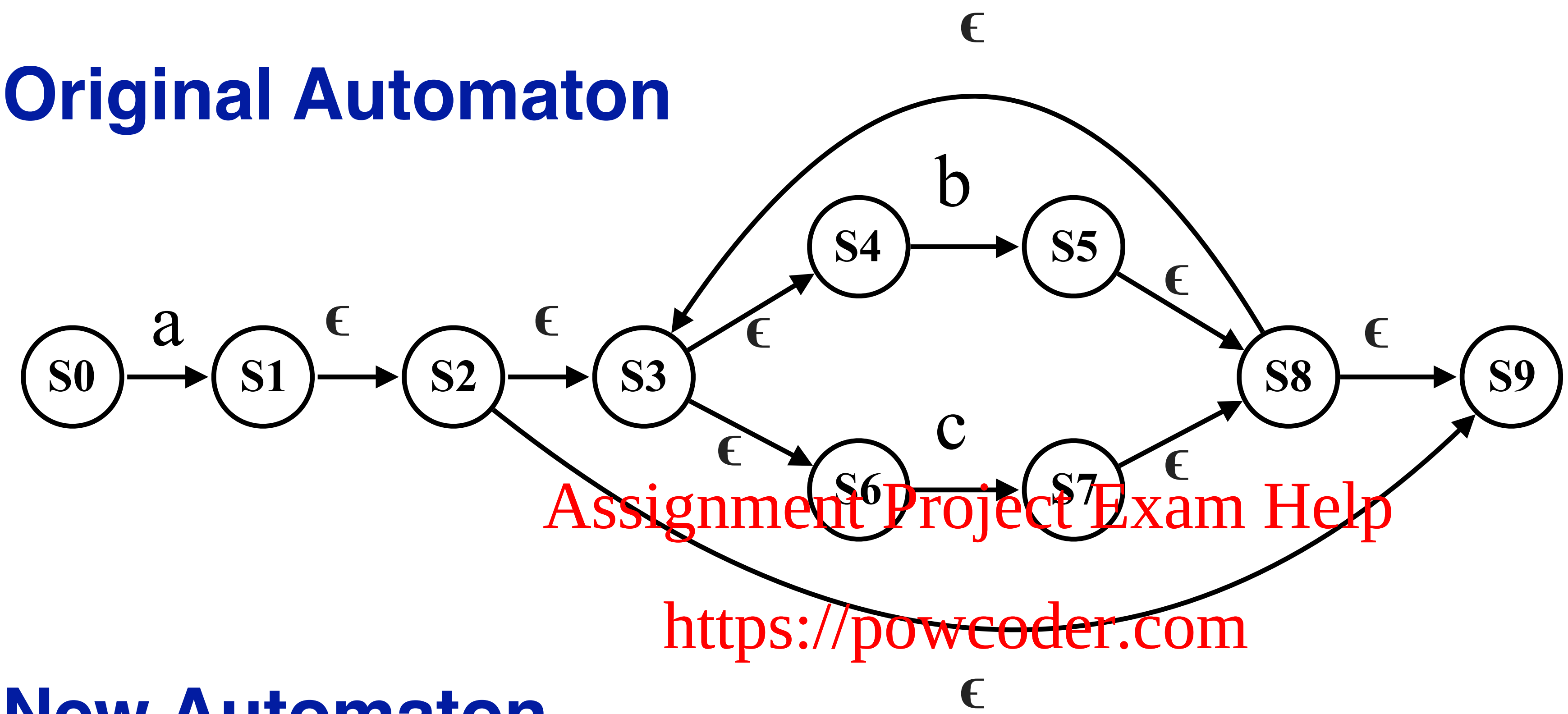


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*
D2	*S5, S8, S3, S9, S4, S6*
D3	*S7, S8, S3, S9, S4, S6*

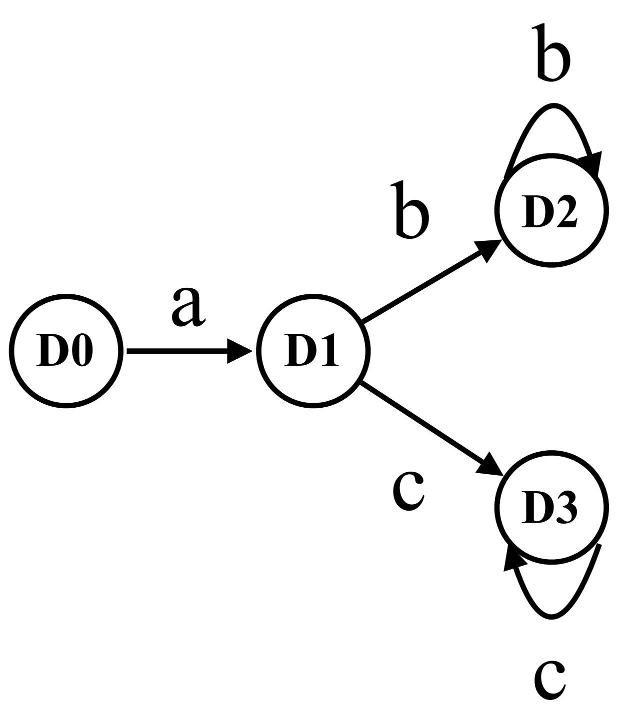
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

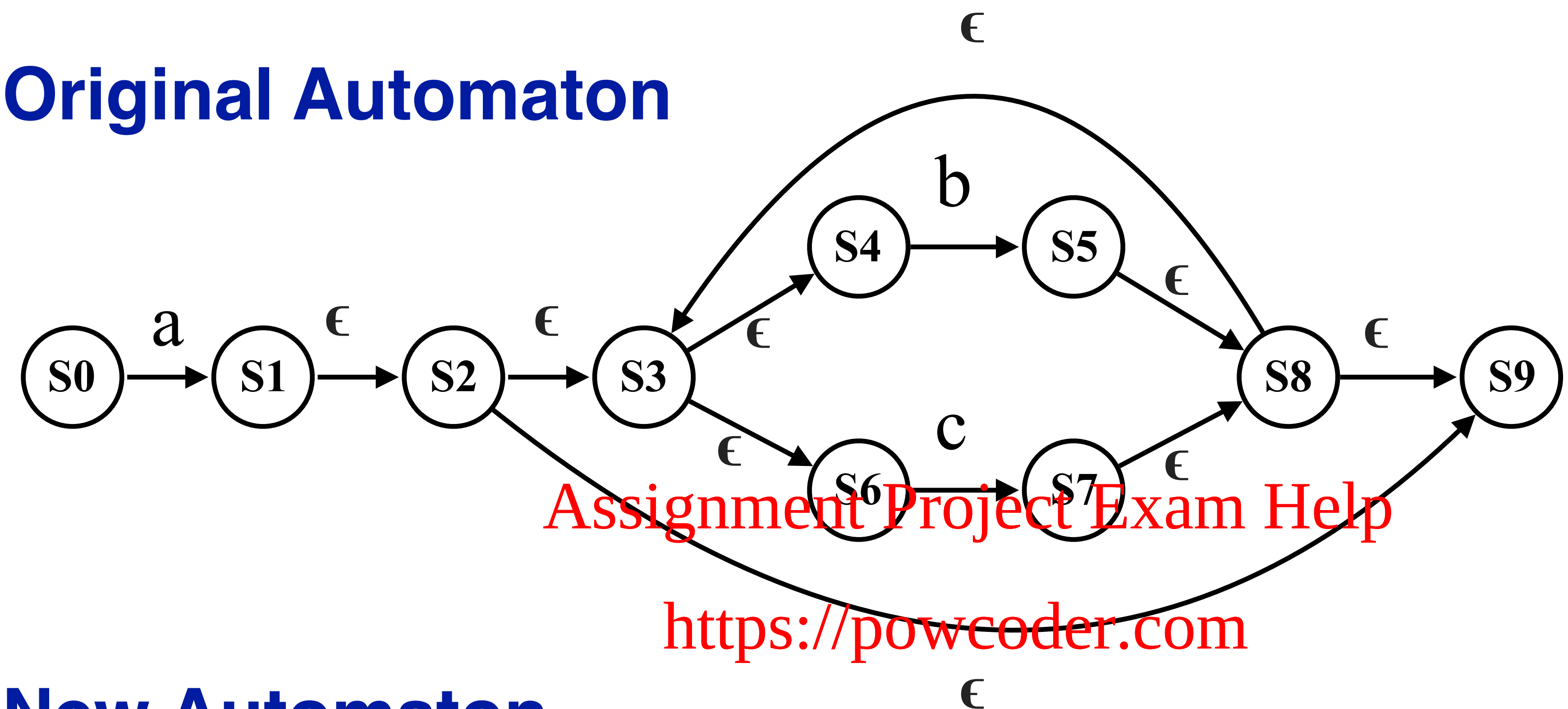


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*
D2	*S5, S8, S3, S9, S4, S6*
D3	*S7, S8, S3, S9, S4, S6*

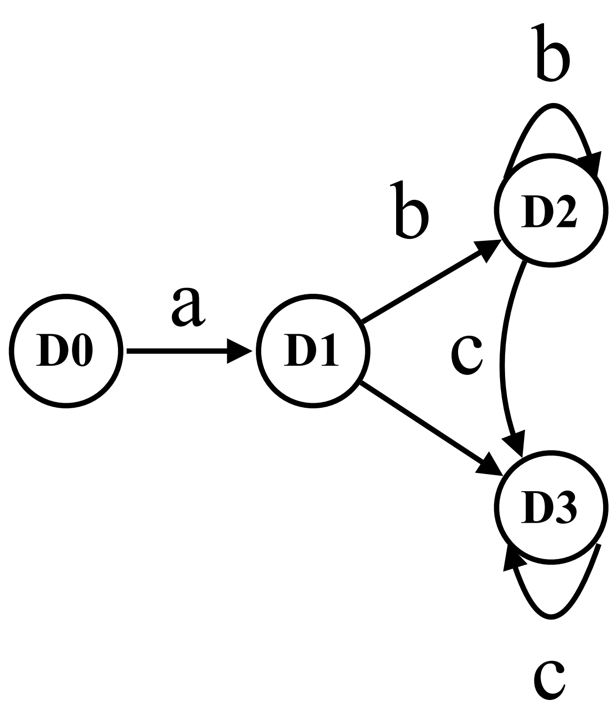
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

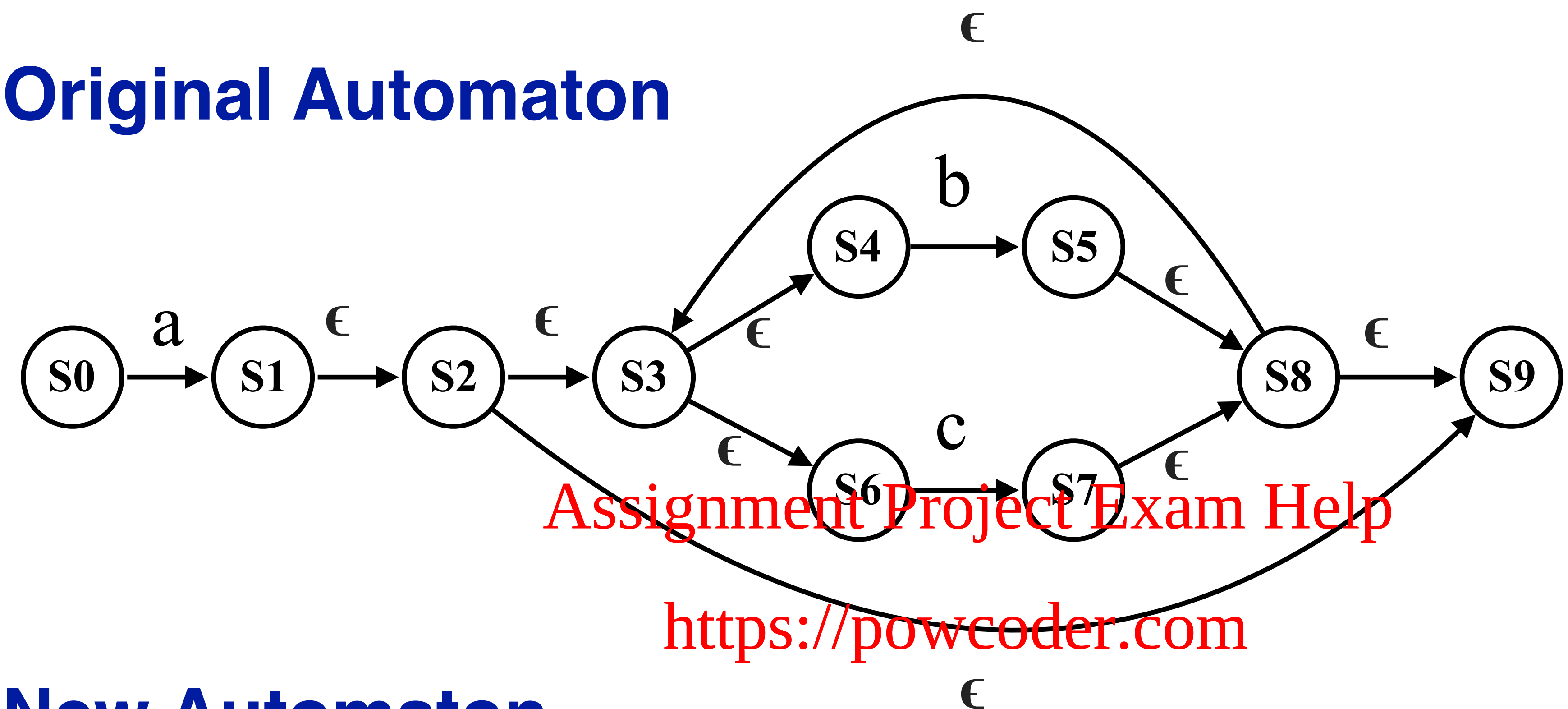


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*
D2	*S5, S8, S3, S9, S4, S6*
D3	*S7, S8, S3, S9, S4, S6*

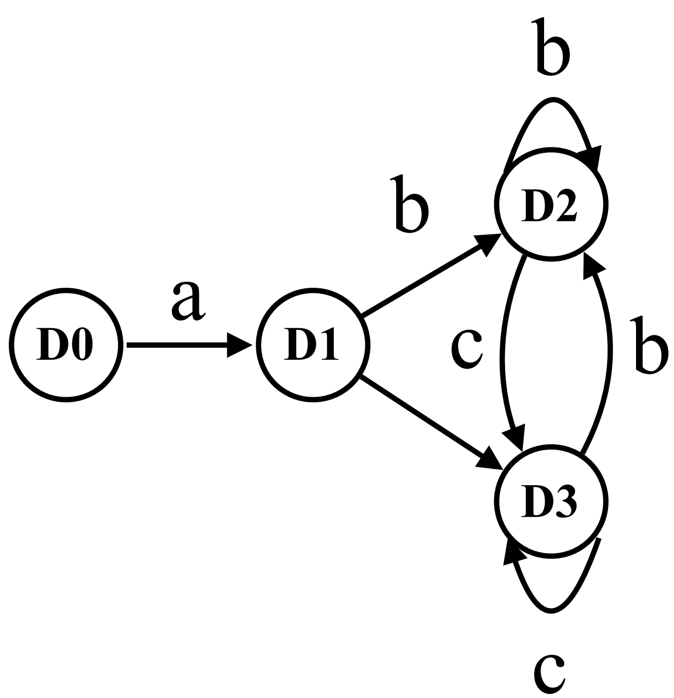
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton

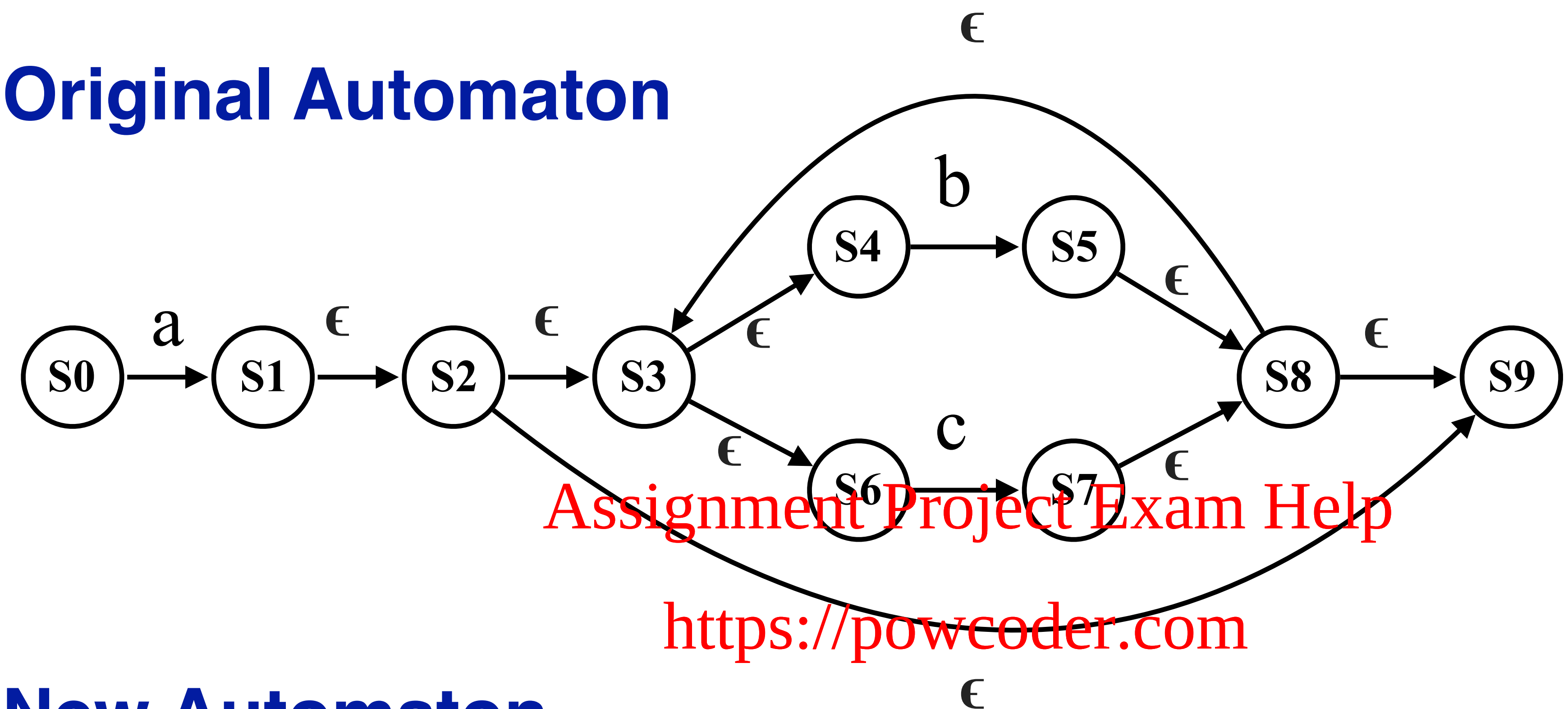


DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*
D2	*S5, S8, S3, S9, S4, S6*
D3	*S7, S8, S3, S9, S4, S6*

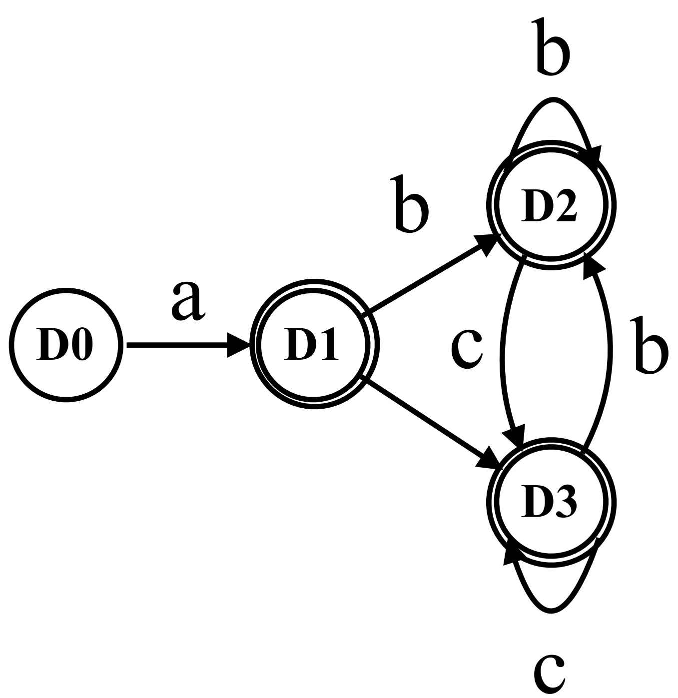
Subset Construction

- Each state in the new one represents a set of states in the original one

Original Automaton



New Automaton



DFA	NFA
D0	*S0*
D1	*S1, S2, S3, S9, S4, S6*
D2	*S5, S8, S3, S9, S4, S6*
D3	*S7, S8, S3, S9, S4, S6*

From RE to Scanner

Classic approach is a three-step method:

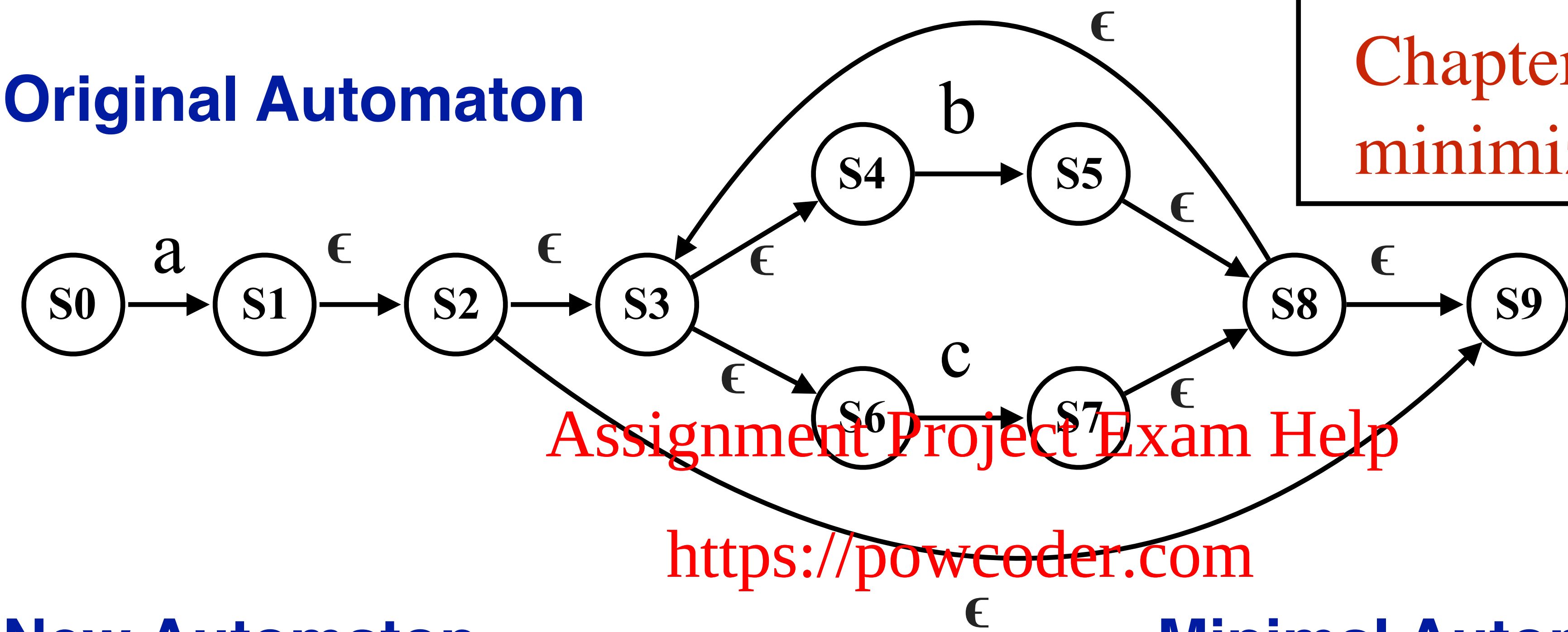
1. Build automata for each piece of the **RE** using a **template**.
   Multiple automata can be pasted using ε-transition.
   This construction is called "**Thompson's construction**"
2. Convert the newly built automaton into a deterministic automaton.
   This construction is called the "**subset construction**"
3. Given the deterministic automaton, minimize the number of states.
   Minimization is a **space optimization**.

DFA Minimization

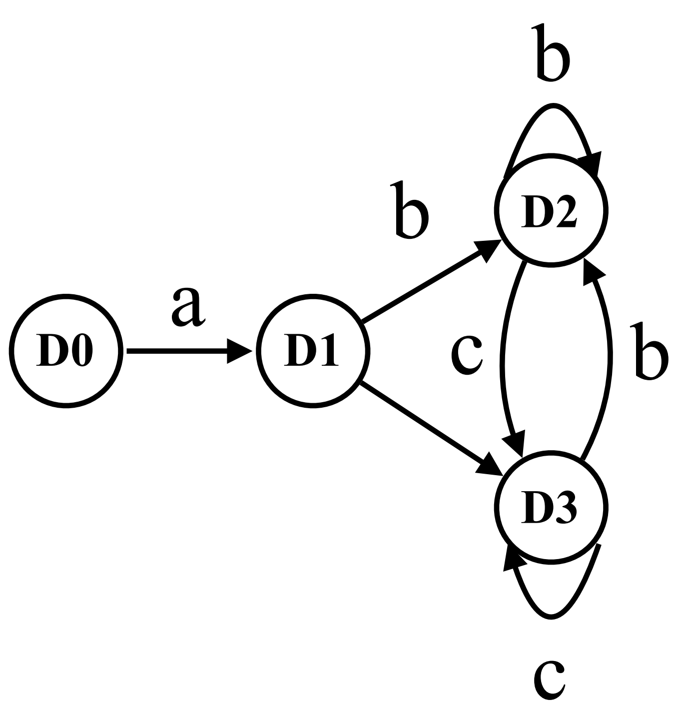
- Discover states that are equivalent in their contexts and replace multiple states with a single one

Read Textbook: Scott, Chapter 2.2.1 for the DFA minimization algorithm.

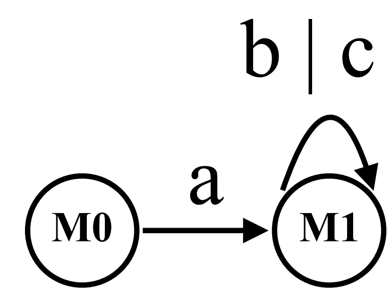
Original Automaton



New Automaton



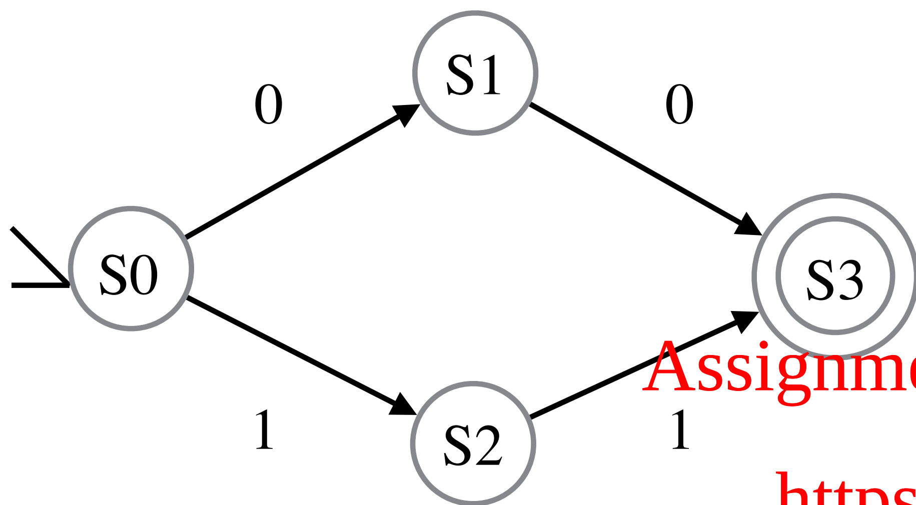
Minimal Automaton



Minimal DFA	Original DFA
M0	*D0*
M1	*D1, D2, D3*

Review: Scanner Implementation

Transitions can be represented using a transition table:



State	*0*	*1*
S0	*S1*	*S2*
S1	*S3*	-
S2	-	*S3*

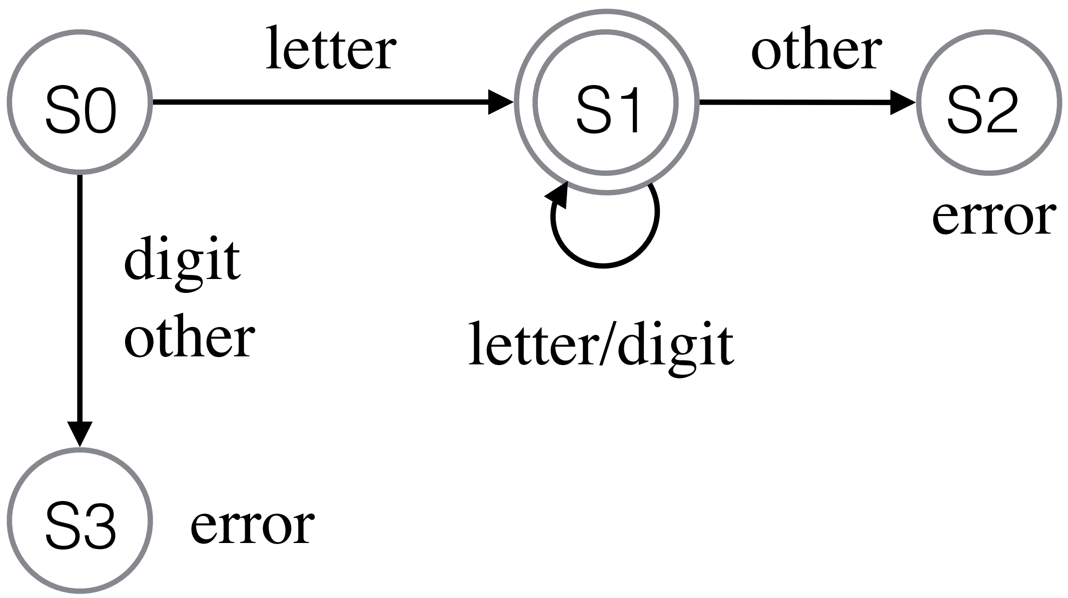
Input

Error

An FSA *accepts/recognizes* an input string **iff** there is some path from start state to a final state such that the labels on the path are that string.

Lack of entry in the table (or no arc for a given character) indicates an *error—reject*.

Review: Code for the scanner

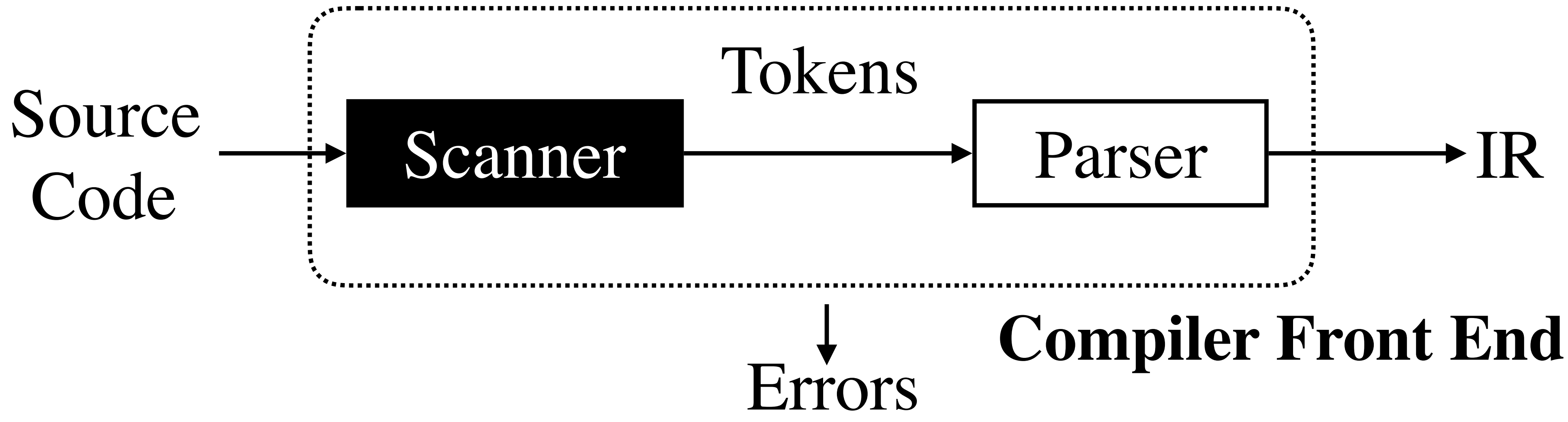


class	S0	S1	S2	S3
letter	S1	S1	—	—
digit	S3	S1	—	—
other	S3	S2	—	—

Implementation:

```
char ← next_char();
state ← S0;
done ← false;
while( not done ) {
    class ← char_class[char];
    state ← next_state[class,state];
    switch(state) {
        case S1:
            /* building an id */
            token_value ← token_value + char;
            char ← next_char();
            if (char == DELIMITER)
                done = true;
            break;
        case S2: /* error state */
        case S3: /* error state */
            token type = error;
            done = true;
            break;
    }
}
return token_type;
```

Compiler Front End

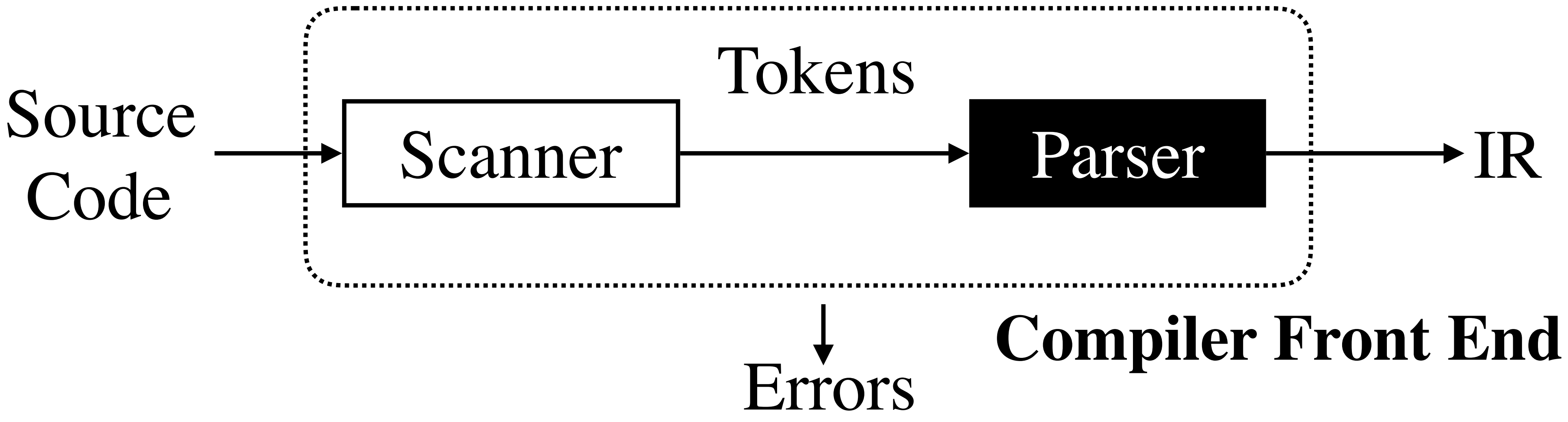


Syntax & semantic analyzer, IR code generator

Front End Responsibilities:

- Recognize legal programs
- Report errors
- Produce intermediate representation

Compiler Front End



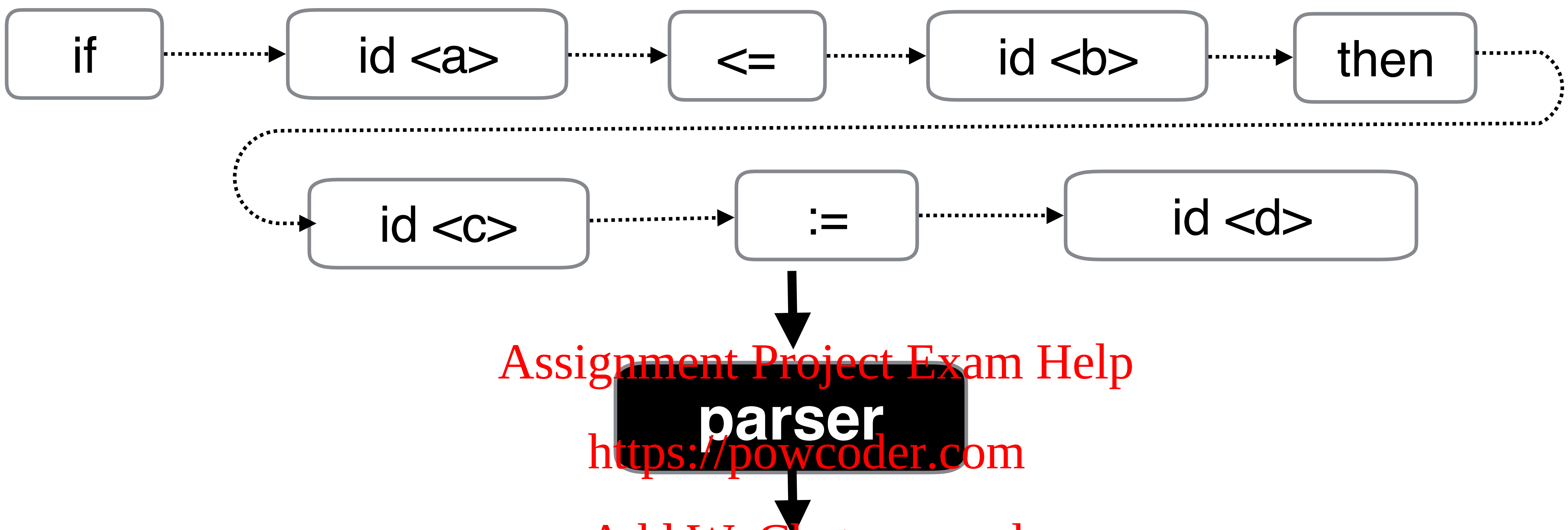
Syntax & semantic analyzer, IR code generator

Front End Responsibilities:

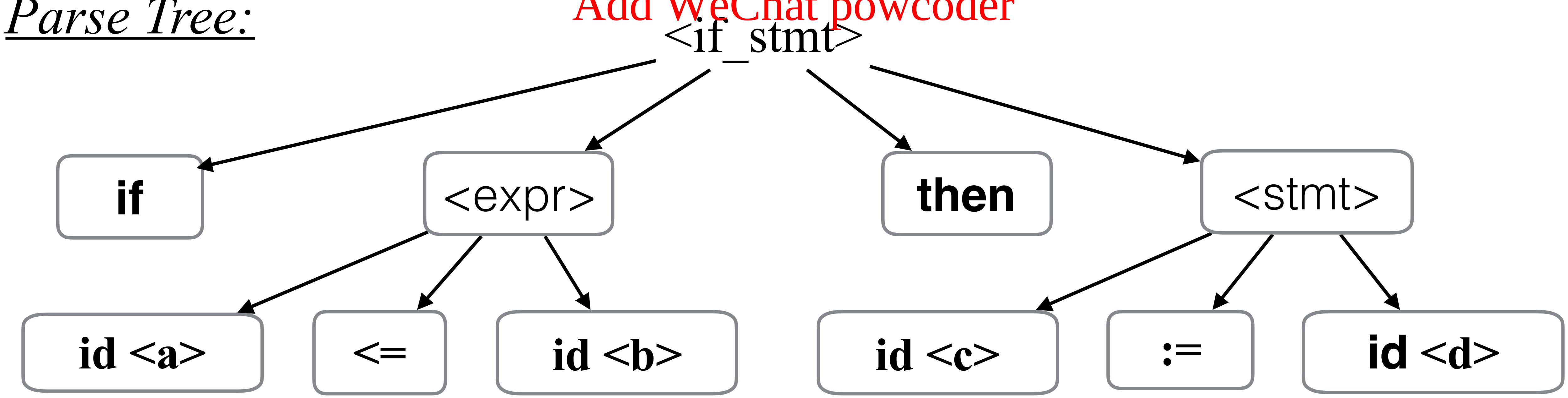
- Recognize legal programs
- Report errors
- Produce intermediate representation

Syntax Analysis (Scott, Chapter 2.1, 2.3)

Token Sequence:



Parse Tree:



Context Free Grammars (CFGs)

- Another formalism to for describing languages
- A CFG $\mathcal{G} = \ < T, N, P, S >$:

  1. A set T of terminal symbols (tokens).
  2. A set N of nonterminal symbols.
  3. A set P production (rewrite) rules.
  4. A special start symbol S.

- The language L($\mathcal{G}$) is the set of sentences of terminal symbols in T* that can be **derived** from the start symbol S:

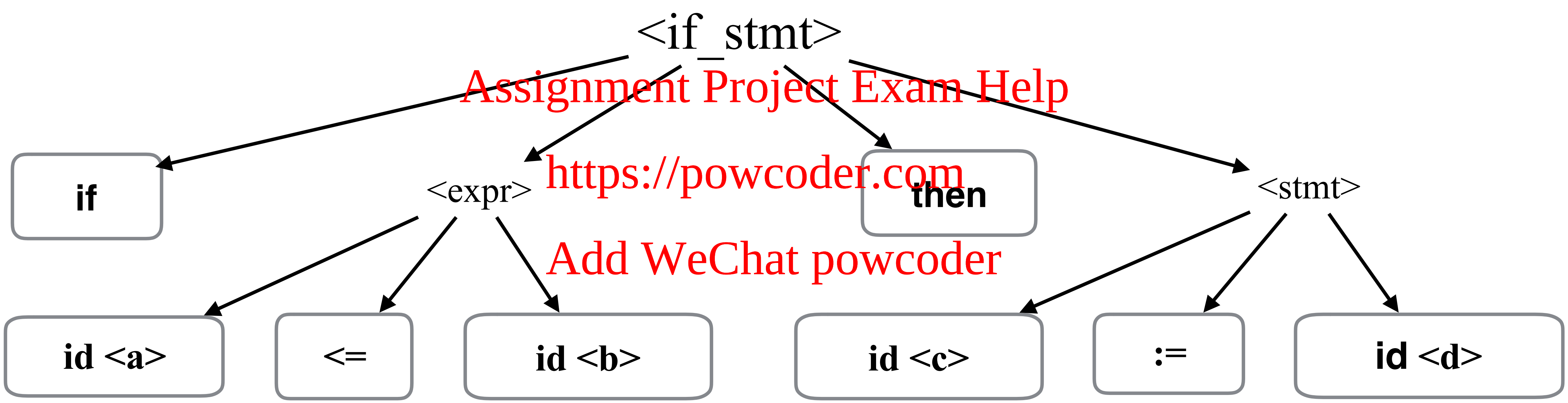
$$L(\mathcal{G}) = \{w \in T^* \mid S \Rightarrow^* w\}$$

An Example of a Partial Context Free Grammar

<if-stmt> ::= **if** <expr> **then** <stmt>

<expr> ::= **id** <= **id**

<stmt> ::= **id** := **id**



Context Free Grammars (CFGs)

- A formalism to for describing languages
- A CFG $\mathcal{G} = \ <\mathbf{T}, \mathbf{N}, \mathbf{P}, \mathbf{S}>$:

  1. A set T of terminal symbols (tokens).
  2. A set N of nonterminal symbols.
  3. A set P production (rewrite) rules.
  4. A special start symbol S.

- The language $L(\mathcal{G})$ is the set of sentences of terminal symbols in T* that can be derived from the start symbol S:

$$L(\mathcal{G}) = \{w \in T^* \mid S \Rightarrow^* w\}$$

CFGs are rewrite systems with restrictions on the form of rewrite (production) rules that can be used.

A Partial Context Free Grammar

...

<if-stmt> ::= **if** <expr> **then** <stmt>
<expr> ::= **id** **<=** **id**
<stmt> ::= **id** **:=** **num**

...

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

Context free grammar

Not a context free grammar

CFGs are rewrite systems with restrictions on the form of rewrite (production) rules that can be used. The left hand side of a production rule can only be **one non-terminal symbol**.

Elements of BNF Syntax

Terminal Symbol:	**Symbol-in-Boldface**
Non-Terminal Symbol:	*Symbol-in-Angle-Brackets*
Production Rule:	Non-Terminal ::= Sequence of Symbols
	or
	Non-Terminal ::= Sequence | Sequence |
	…

Example:

…

<if-stmt> ::= **if** <expr> **then** <stmt>

<expr> ::= **id** **<=** **id**

<stmt> ::= **id** **:=** **num**

How a BNF Grammar Describes a Language

A sentence is a sequence of terminal symbols (tokens).

The language L($\mathcal{G}$) of a BNF grammar $\mathcal{G}$ is the set of sentences generated using the grammar:

- Begin with start symbol.
- Iteratively replace non-terminals with terminals according to the rules (rewrite system).

This replacing process is called a derivation ($\Rightarrow$).
Zero or multiple derivation steps are written as $\Rightarrow$.*

Formally, $L(\mathcal{G}) = \{w \in T^* \mid S \Rightarrow^* w\}$

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), in another word, can X2 := 0 be derived in $\mathcal{G}$?

Start Symbol : <stmt>

↓

⋮ ?

↓

X2 := 0

Grammar $\mathcal{G}$:

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

In which order to apply the rules?

Typically, there are three options:

leftmost ($\Rightarrow_L$), rightmost ($\Rightarrow_R$), any ($\Rightarrow$)

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 $\in$ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation	rule
<stmt> $\Rightarrow_L$	
X2 := 0	

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 $\in$ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation		rule
<stmt>	$\Rightarrow_L$	6
<identifier> := 0	$\Rightarrow_L$	
X2 := 0		

1. $<$ letter $>$ ::= A \| B \| C \| … \| Z
2. $<$ digit $>$ ::= 0 \| 1 \| 2 \| … \| 9
3. $<$ identifier $>$ ::= $<$ letter $>$ \|
4. $<$ identifier $>$ $<$ letter $>$ \|
5. $<$ identifier $>$ $<$ digit $>$
6. $<$ stmt $>$ ::= $<$ identifier $>$:= 0

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 $\in$ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation		rule
<stmt>	$\Rightarrow_L$	6
<identifier> := 0	$\Rightarrow_L$	
X2 := 0		

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation		rule
<stmt>	$\Rightarrow_L$	6
<identifier> := 0	$\Rightarrow_L$	5
<identifier> <digit> := 0	$\Rightarrow_L$	
X2 := 0		

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation		rule
<stmt>	$\Rightarrow_L$	6
<identifier> := 0	$\Rightarrow_L$	5
<identifier> <digit> := 0	$\Rightarrow_L$	
X2 := 0		

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 $\in$ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation	rule
<stmt> $\Rightarrow_L$	6
<identifier> := 0 $\Rightarrow_L$	5
<identifier> <digit> := 0 $\Rightarrow_L$	3
<letter> <digit> := 0 $\Rightarrow_L$	
X2 := 0	

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 $\in$ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation		rule
<stmt>	$\Rightarrow_L$	6
<identifier> := 0	$\Rightarrow_L$	5
<identifier> <digit> := 0	$\Rightarrow_L$	3
<letter> <digit> := 0	$\Rightarrow_L$	
X2 := 0		

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation	rule
<stmt> $\Rightarrow_L$	6
<identifier> := 0 $\Rightarrow_L$	5
<identifier> <digit> := 0 $\Rightarrow_L$	3
<letter> <digit> := 0 $\Rightarrow_L$	1
X <digit> := 0 $\Rightarrow_L$	
X2 := 0	

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 $\in$ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation		rule
<stmt>	$\Rightarrow_L$	6
<identifier> := 0	$\Rightarrow_L$	5
<identifier> <digit> := 0	$\Rightarrow_L$	3
<letter> <digit> := 0	$\Rightarrow_L$	1
X <digit> := 0	$\Rightarrow_L$	
X2 := 0		

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 $\in$ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation	rule
<stmt> $\Rightarrow_L$	6
<identifier> := 0 $\Rightarrow_L$	5
<identifier> <digit> := 0 $\Rightarrow_L$	3
<letter> <digit> := 0 $\Rightarrow_L$	1
X <digit> := 0 $\Rightarrow_L$	2
X2 := 0	

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 $\in$ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation		rule
<stmt>	$\Rightarrow_L$	6
<identifier> := 0	$\Rightarrow_L$	5
<identifier> <digit> := 0	$\Rightarrow_L$	3
<letter> <digit> := 0	$\Rightarrow_L$	1
X <digit> := 0	$\Rightarrow_L$	2
X2 := 0		

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

leftmost derivation	rule
<stmt> $\Rightarrow_L$	6
<identifier> := 0 $\Rightarrow_L$	5
<identifier> <digit> := 0 $\Rightarrow_L$	3
<letter> <digit> := 0 $\Rightarrow_L$	1
X <digit> := 0 $\Rightarrow_L$	2
X2 := 0	

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

Next Lecture

Things to do:

- Read Scott, Chapter 2.2 - 2.5 (skip 2.3.3 bottom-up Parsing)