CS311—Computational Structures

Regular Languages and Regular Grammars

Lecture 6

What we know so far:

- RLs are closed under product, union and *
- Every RL can be written as a RE, and every RE represents a RL
- Every RL can be recognized by a NFA
  - and we know how to build it
- NFAs and DFA have the same “power”
- Every NFA can be turned in to a DFA
  - “the subset construction”

What's Next?

- How to turn a FSA into a regular grammar
  - and vice-versa
- Minimal-state DFAs
  - Myhill-Nerode Theorem
    - Language indistinguishability

Phrase-Structure Grammars

- A grammar is a set of **rules** for transforming strings
  - Strings can involve **variables** and **terminal** symbols
  - $S \rightarrow abcT$
- We **derive** a string of terminals by repeatedly applying rules beginning from a designated **start variable** (often S)
  - The **language** defined by a grammar is the set of strings that can be derived

Phrase-Structure Grammars

- A grammar is a set of **rules** for transforming strings
  - Strings can involve **variables** and **terminal** symbols
  - S → abcT
- We **derive** a string of terminals by repeatedly applying rules beginning from a designated **start variable** (often S)
  - The **language** defined by a grammar is the set of strings that can be derived

Phrase-Structure Grammars

- A grammar is a set of **rules** for transforming strings
  - Strings can involve **variables** and **terminal** symbols
  - S → abcT

Variables

- We **derive** a string of terminals by repeatedly applying rules beginning from a designated **start variable** (often S)
  - The **language** defined by a grammar is the set of strings that can be derived

Phrase-Structure Grammars

- A grammar is a set of **rules** for transforming strings
  - Strings can involve **variables** and **terminal** symbols
  - S → abcT
- We **derive** a string of terminals by repeatedly applying rules beginning from a designated **start variable** (often S)
  - The **language** defined by a grammar is the set of strings that can be derived

Phrase-Structure Grammars

- A grammar is a set of **rules** for transforming strings
  - Strings can involve **variables** and **terminal** symbols
  - S → abcT
- We **derive** a string of terminals by repeatedly applying rules beginning from a designated **start variable** (often S)
  - The **language** defined by a grammar is the set of strings that can be derived

Phrase-Structure Grammars

- A grammar is a set of **rules** for transforming strings
  - Strings can involve **variables** and **terminal** symbols
  - S → abcT

Terminal symbols

- We **derive** a string of terminals by repeatedly applying rules beginning from a designated **start variable** (often S)
  - The **language** defined by a grammar is the set of strings that can be derived

Portland State UNIVERSITY

Regular Grammars

Hein Section 11.4.1

- What's a Regular Grammar?
  - A particular kind of grammar in which all the productions have one of these forms:

    $S \rightarrow \varepsilon$ $\quad$ $S \rightarrow w$ $\quad$ $S \rightarrow T$ $\quad$ $S \rightarrow wT$

  - w is a *sequence* of terminal symbols
  - at most one variable can appear on the rhs, and it *must* be on the right.
- Examples:

  $S \rightarrow abcY$ $\quad$ ~~$Y \rightarrow aZa$~~ $\quad$ ~~$S \rightarrow AB$~~

Portland State University

Examples

Examples

Portland State University

Examples

a*

Portland State University

Examples

a*	$S \rightarrow \varepsilon \mid aS$

Portland State University

Examples

a*	$S \rightarrow \varepsilon \mid aS$
a+b	

Examples

a*	$S \to \varepsilon \mid aS$
a+b	$S \to a \mid b$

Portland State University

Examples

a*	S → ε | aS
a+b	S → a | b
(a+b)*	

Portland State University

Examples

a*	$S \rightarrow \varepsilon \mid aS$
a+b	$S \rightarrow a \mid b$
(a+b)*	$S \rightarrow \varepsilon \mid aS \mid bS$

Portland State University

Examples

a*	$S \to \varepsilon \mid aS$
a+b	$S \to a \mid b$
(a+b)*	$S \to \varepsilon \mid aS \mid bS$
a* + b*	

Portland State UNIVERSITY

Examples

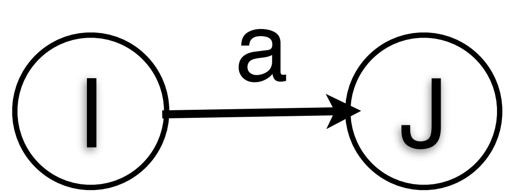
a*	$S \rightarrow \varepsilon \mid aS$
a+b	$S \rightarrow a \mid b$
(a+b)*	$S \rightarrow \varepsilon \mid aS \mid bS$
a* + b*	$S \rightarrow A \mid B$ $A \rightarrow \varepsilon \mid aA$ $B \rightarrow \varepsilon \mid bB$

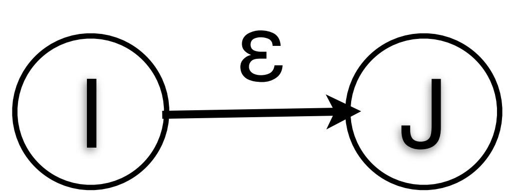
Languages and Grammars

- Any regular language has a regular grammar
- Any regular grammar generates a regular language

From NFA to Regular Grammar

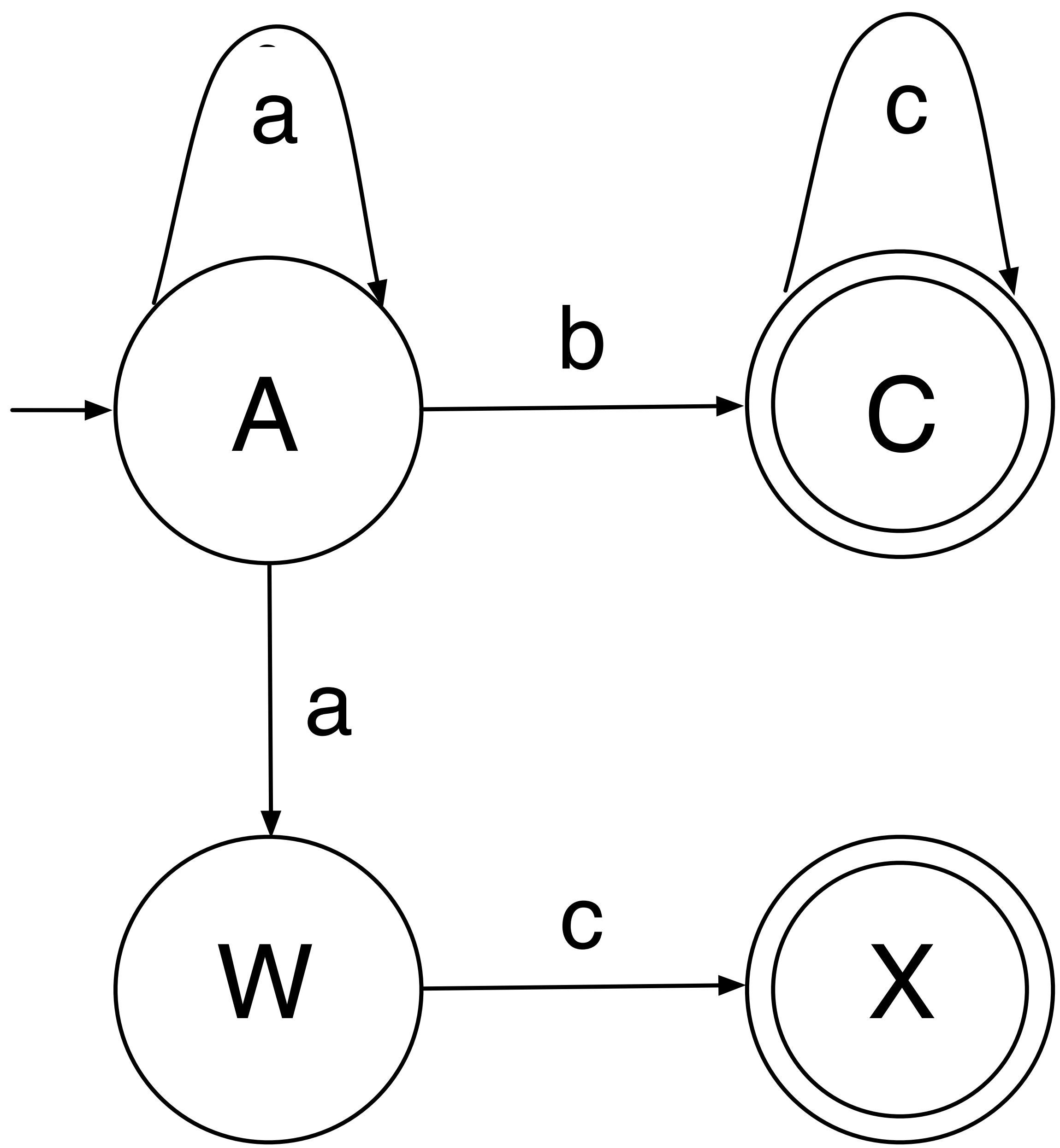
Hein Algorithm 11.11

1. Rename the states Q to a set of upper-case letters
2. The start symbol of the grammar is the name of the start state q_0.
3. For each transition 

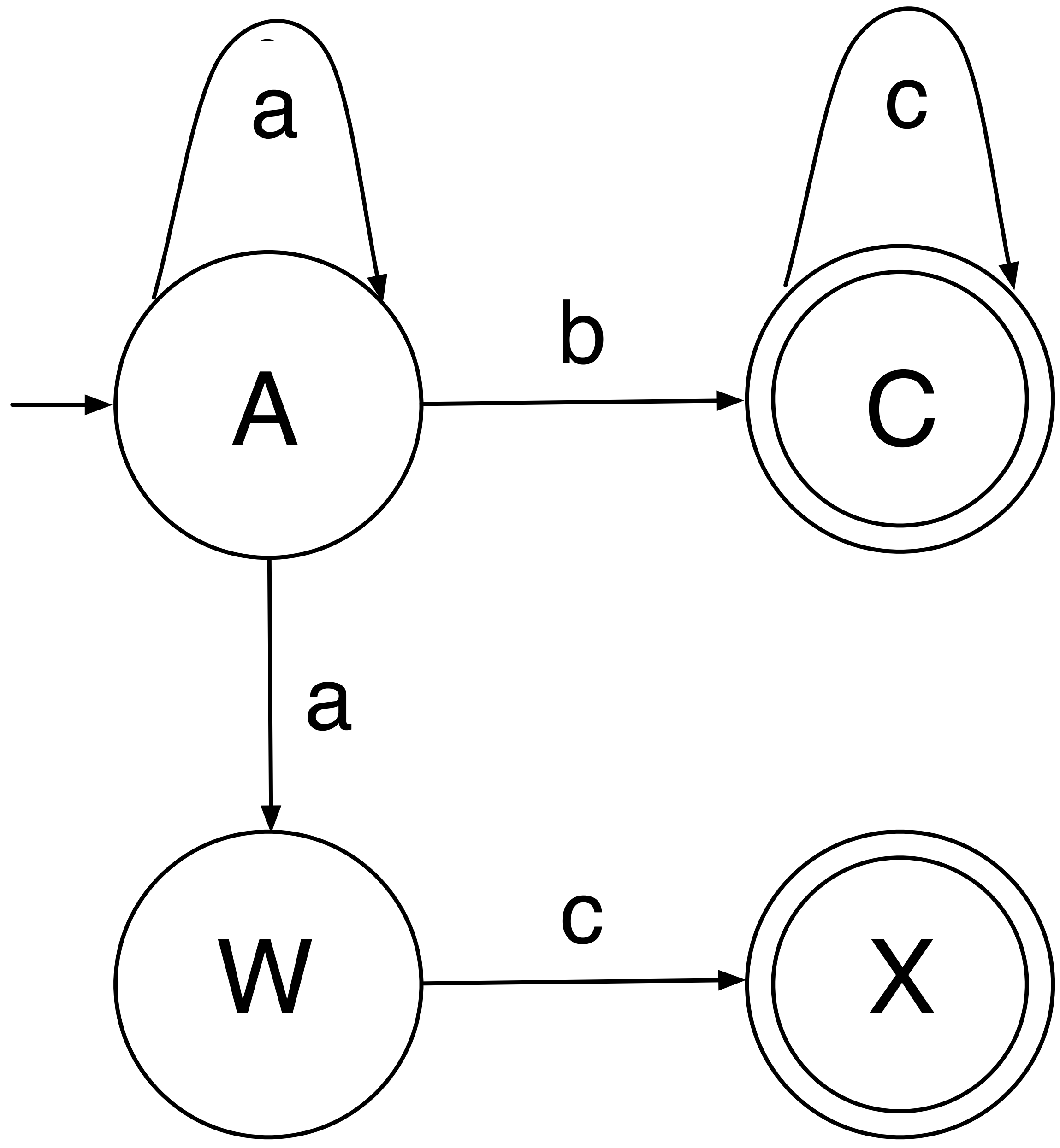
, create the production I→aJ.
4. For each transition 

, create the production I→J.
5. For each final state K, create the production K→ε.

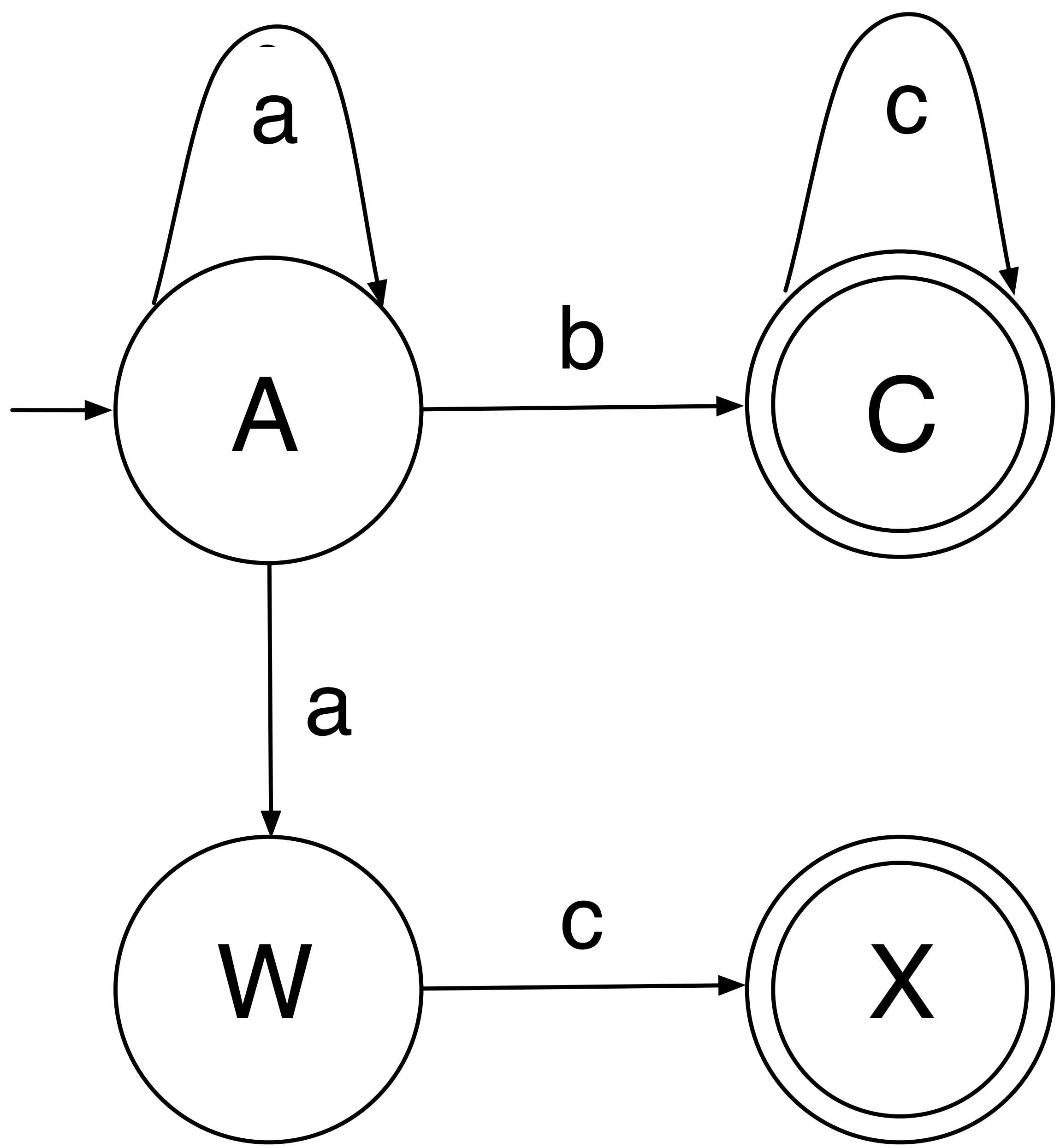
Example



Example

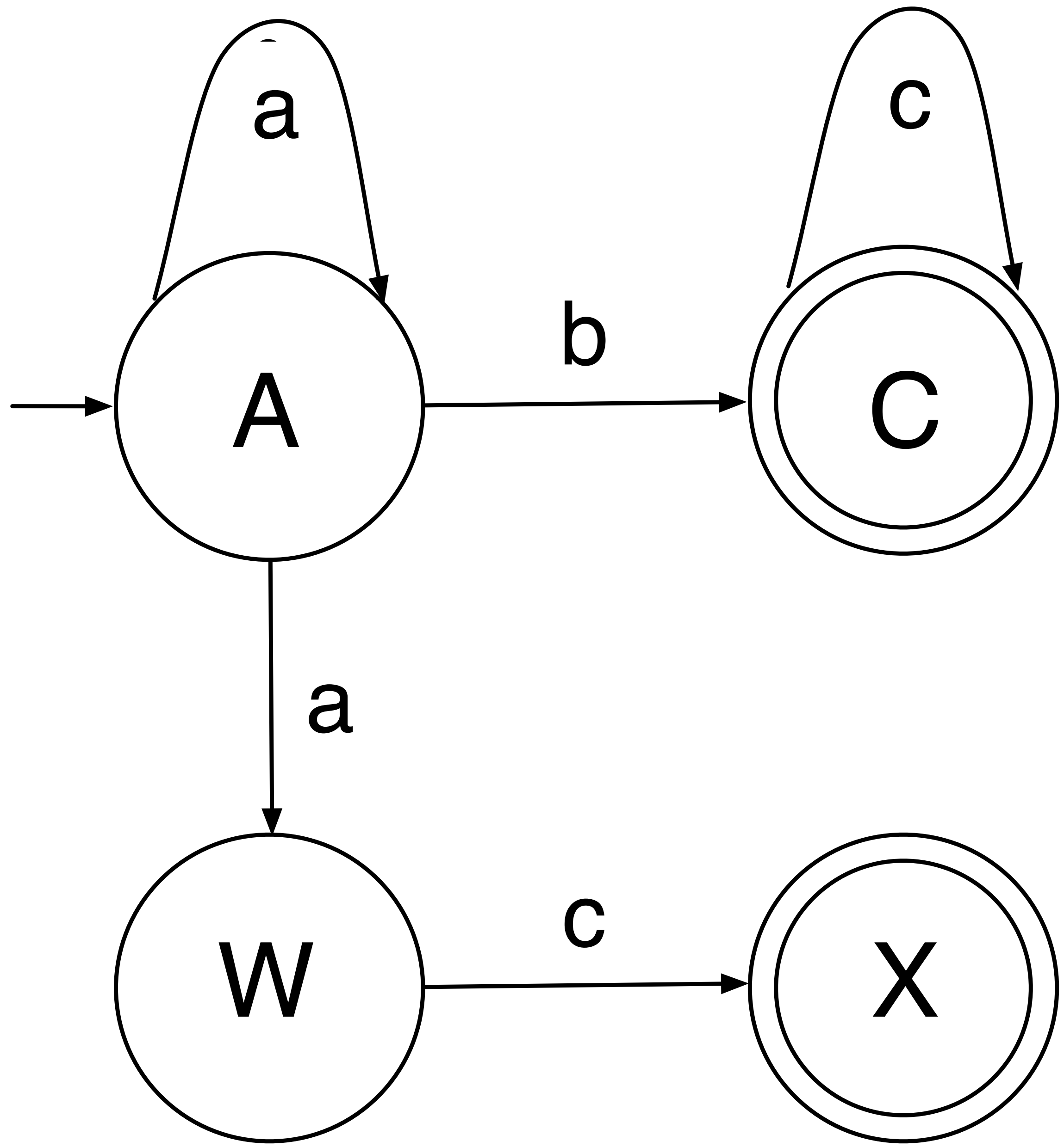


Example



A→aA

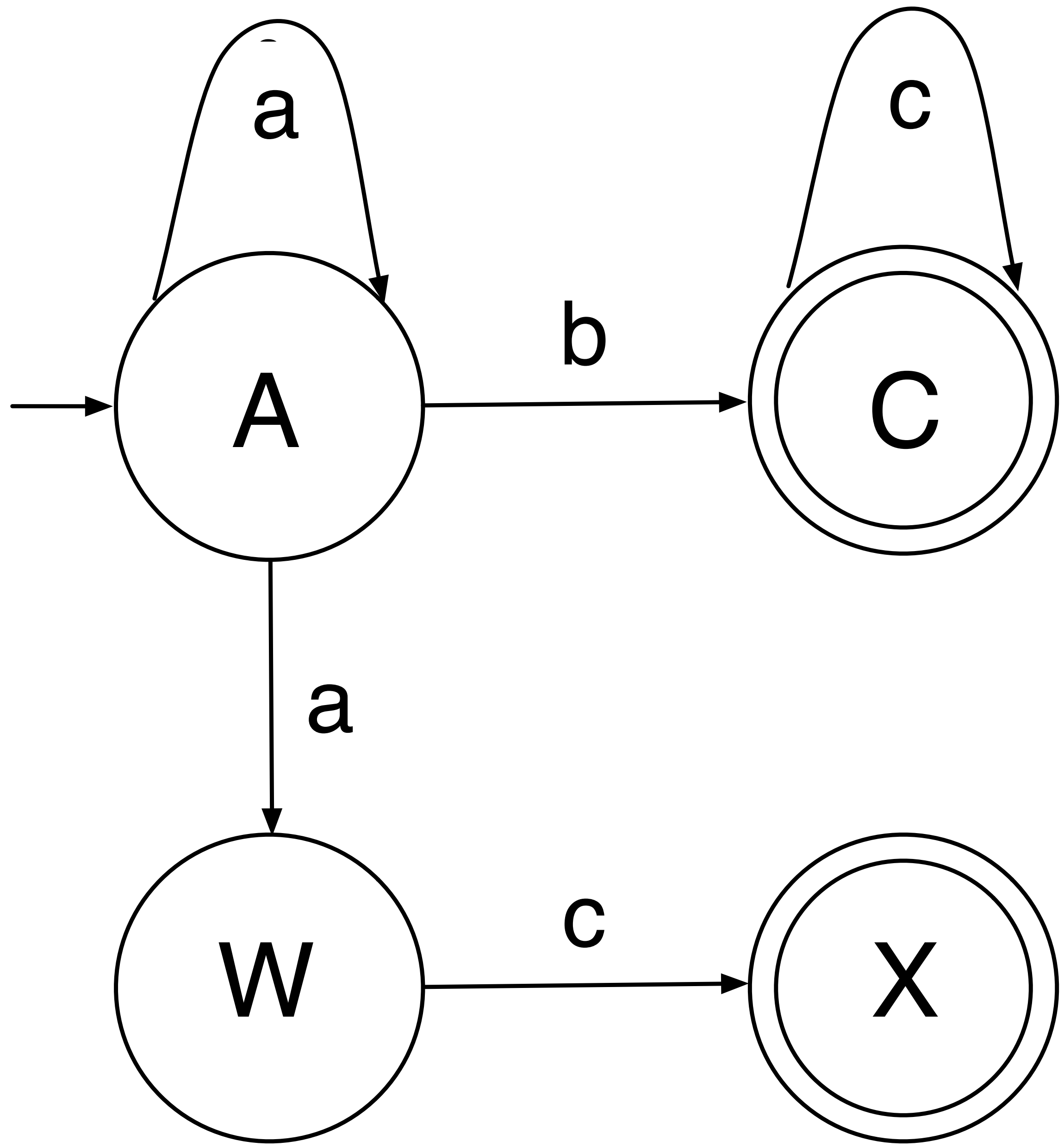
Example



$A \rightarrow aA$

$A \rightarrow bC$

Example



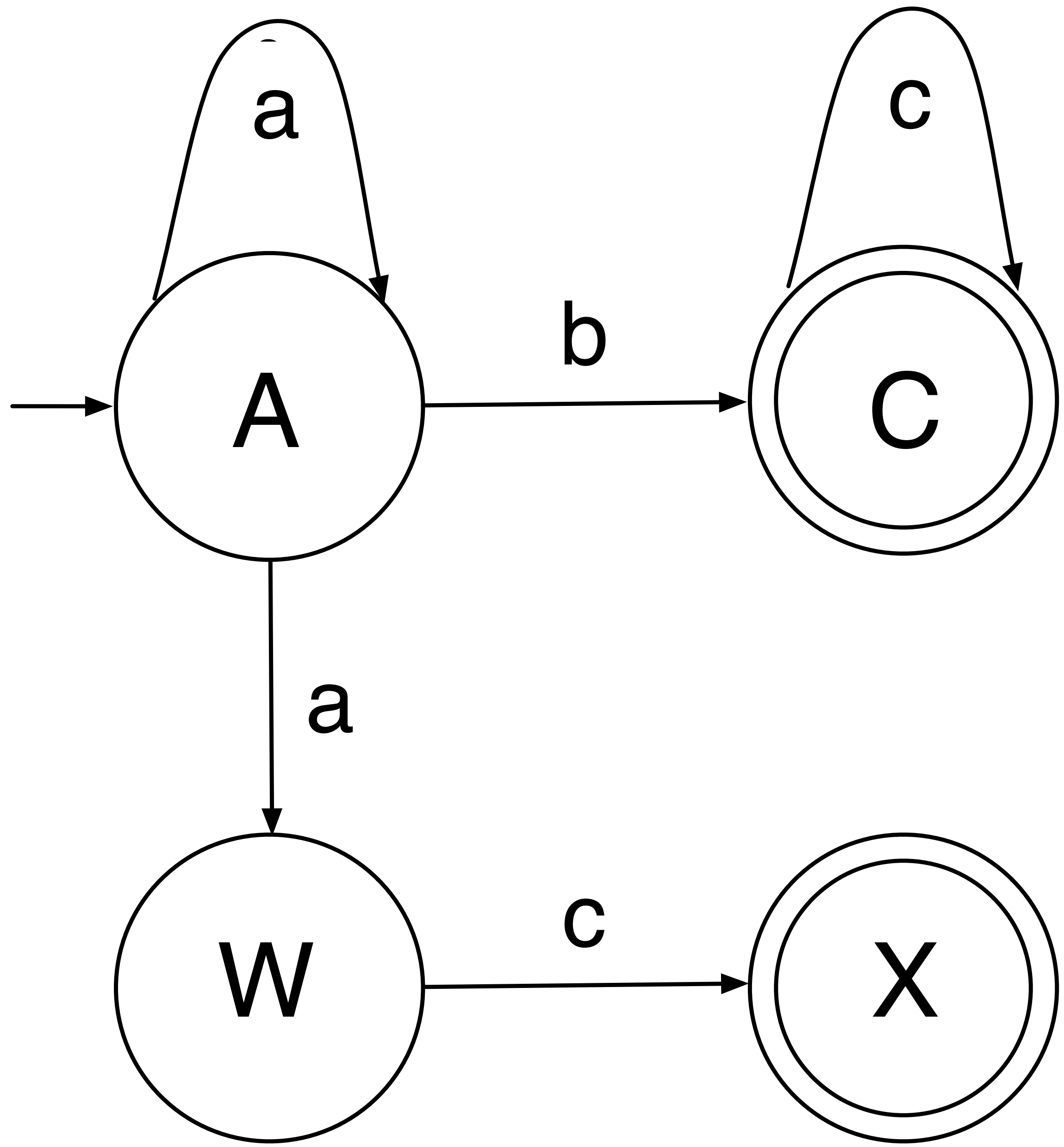
$A \rightarrow aA$

$A \rightarrow bC$

$A \rightarrow aW$

Portland State University

Example



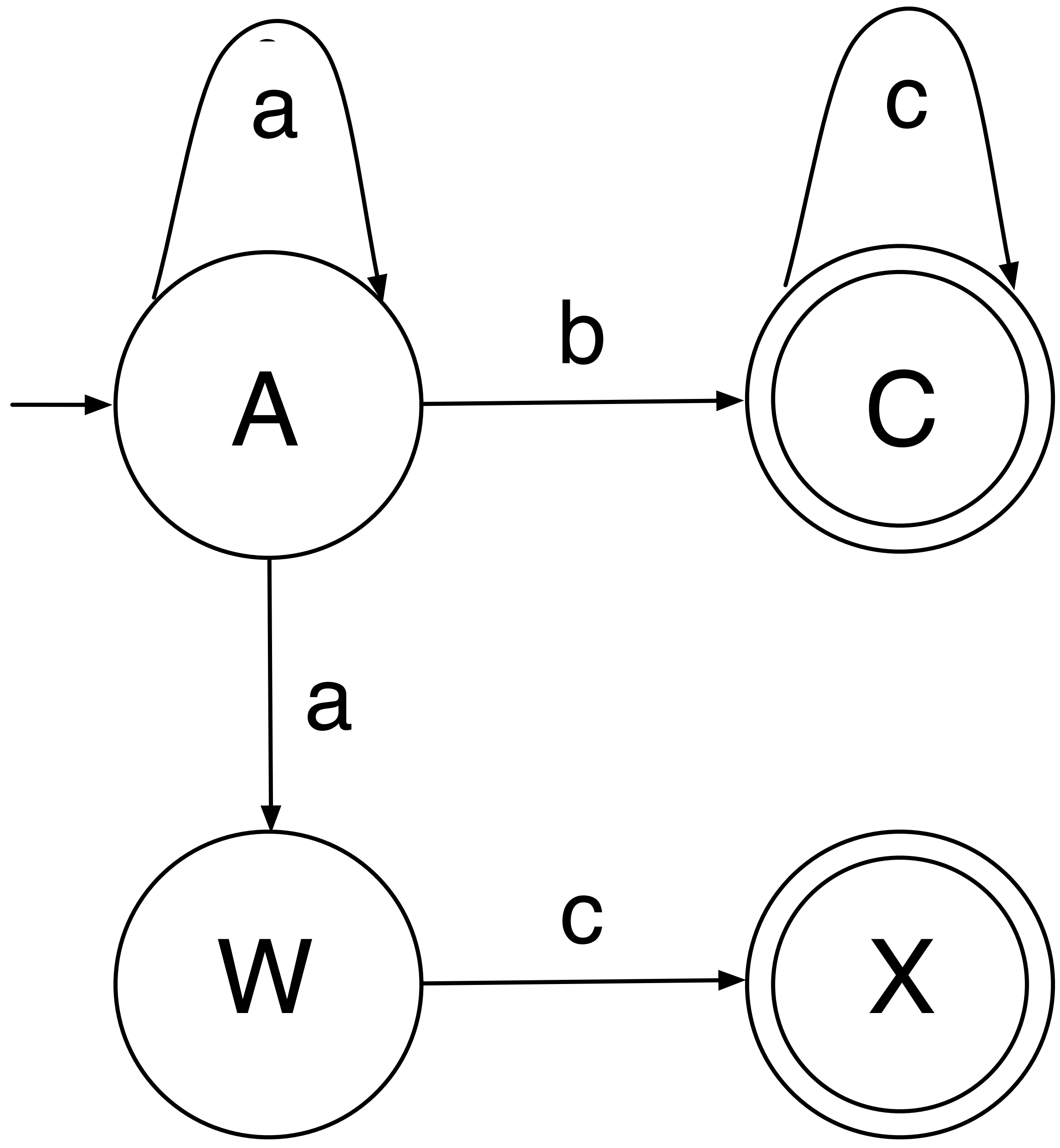
$A \rightarrow aA$

$A \rightarrow bC$

$A \rightarrow aW$

$C \rightarrow cC$

Example



A→aA

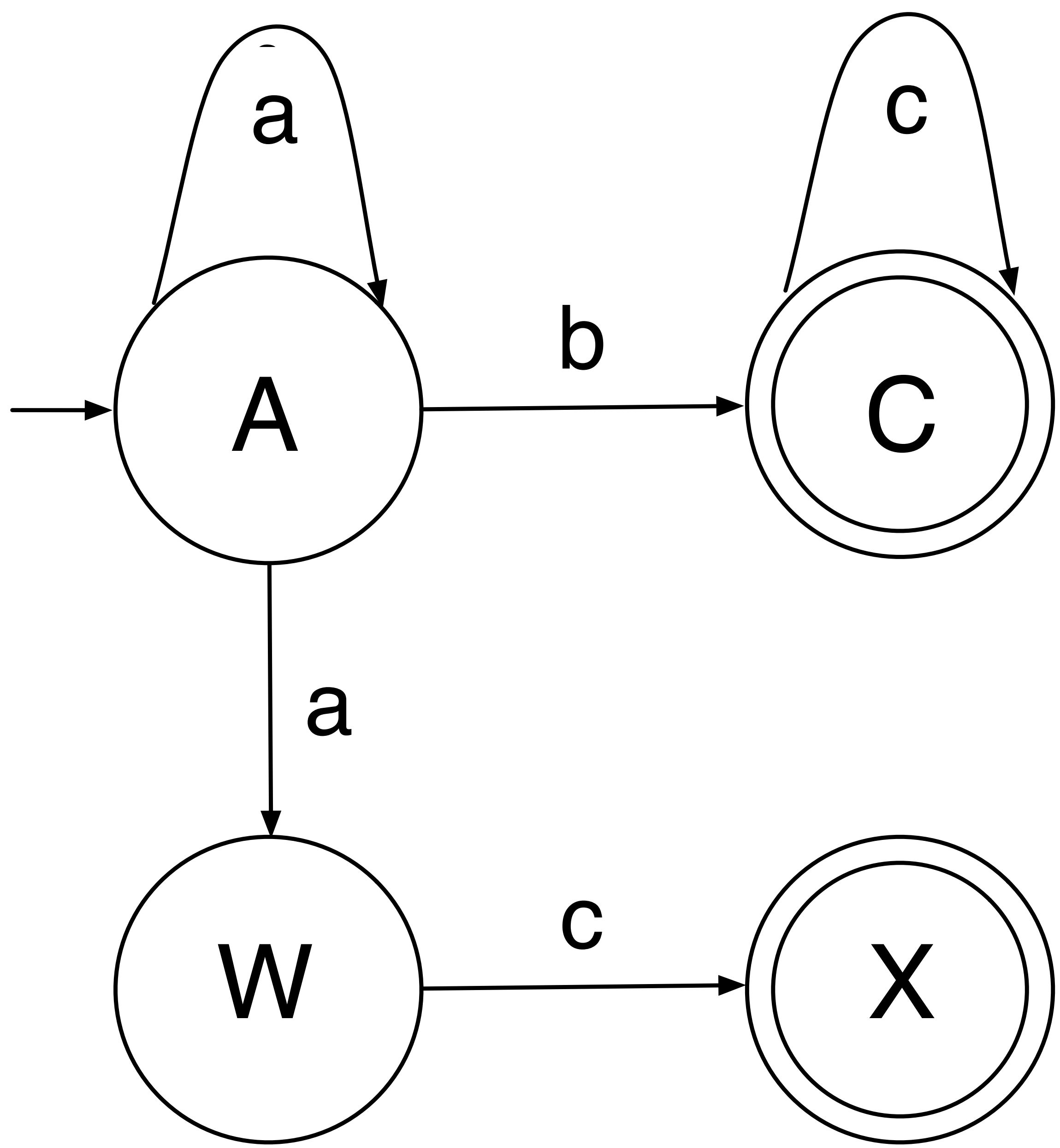
A→bC

A→aW

C→cC

C→ε

Example



A→aA

A→bC

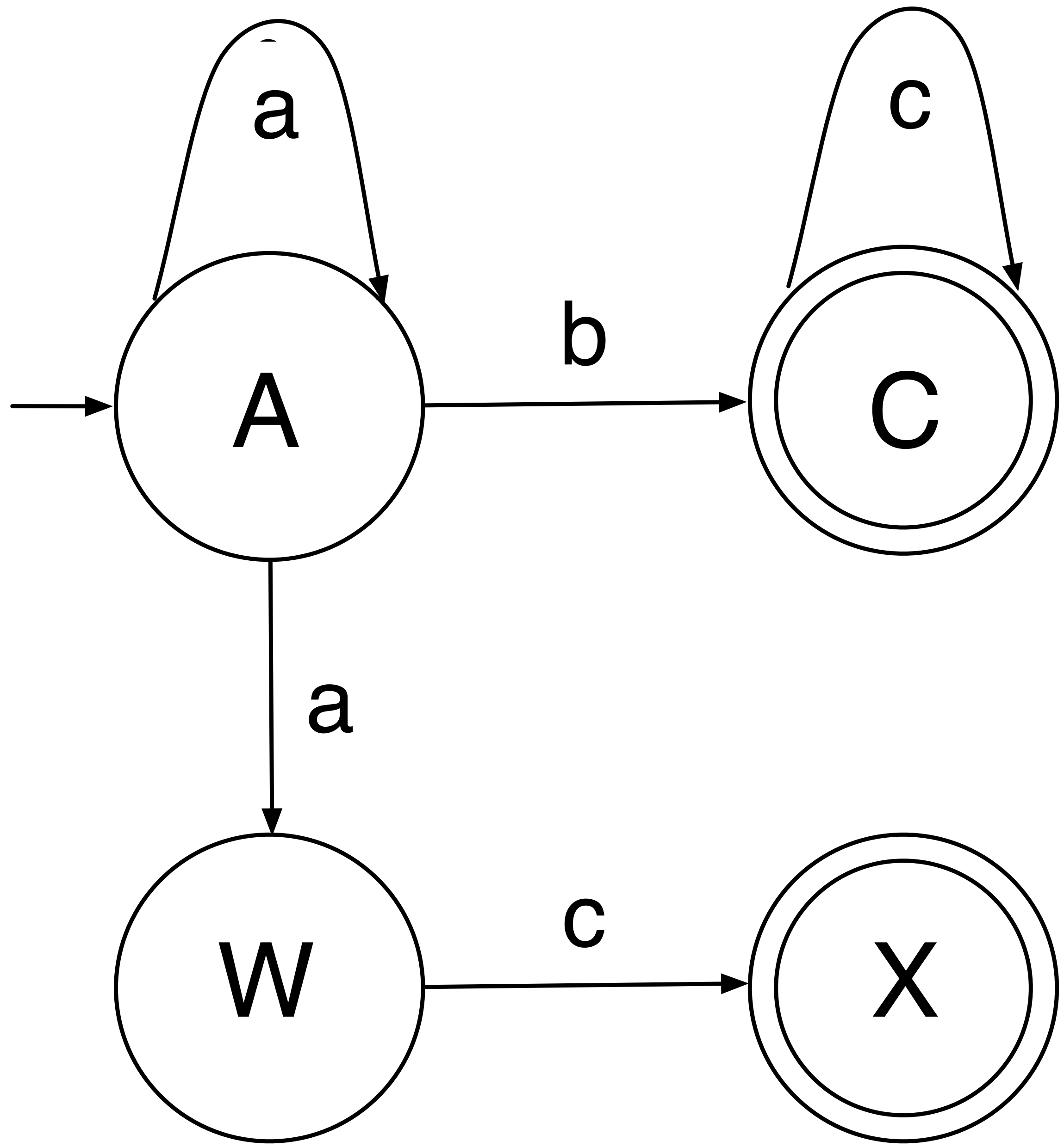
A→aW

C→cC

C→ε

W→cX

Example



$A \rightarrow aA$

$A \rightarrow bC$

$A \rightarrow aW$

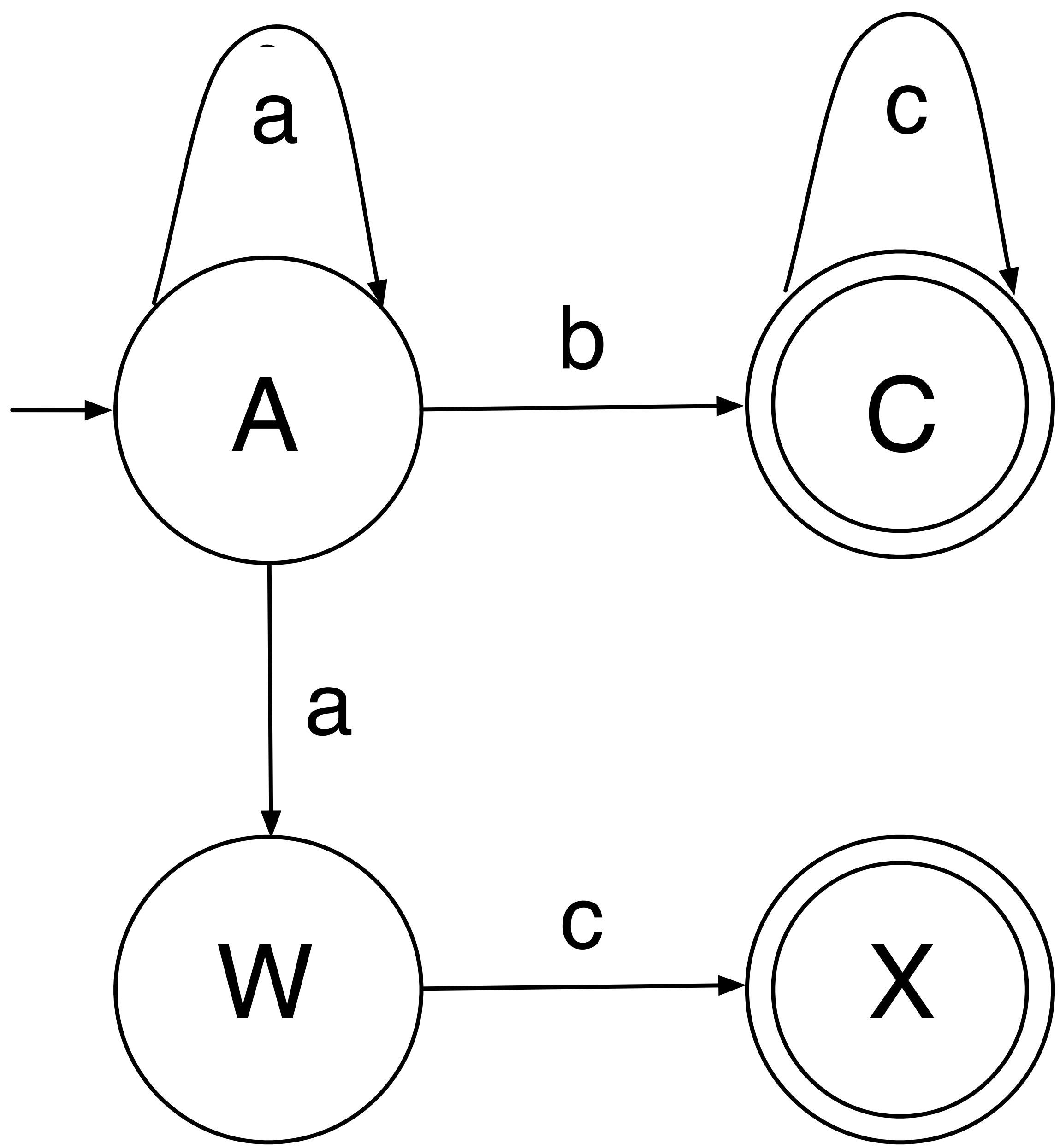
$C \rightarrow cC$

$C \rightarrow \varepsilon$

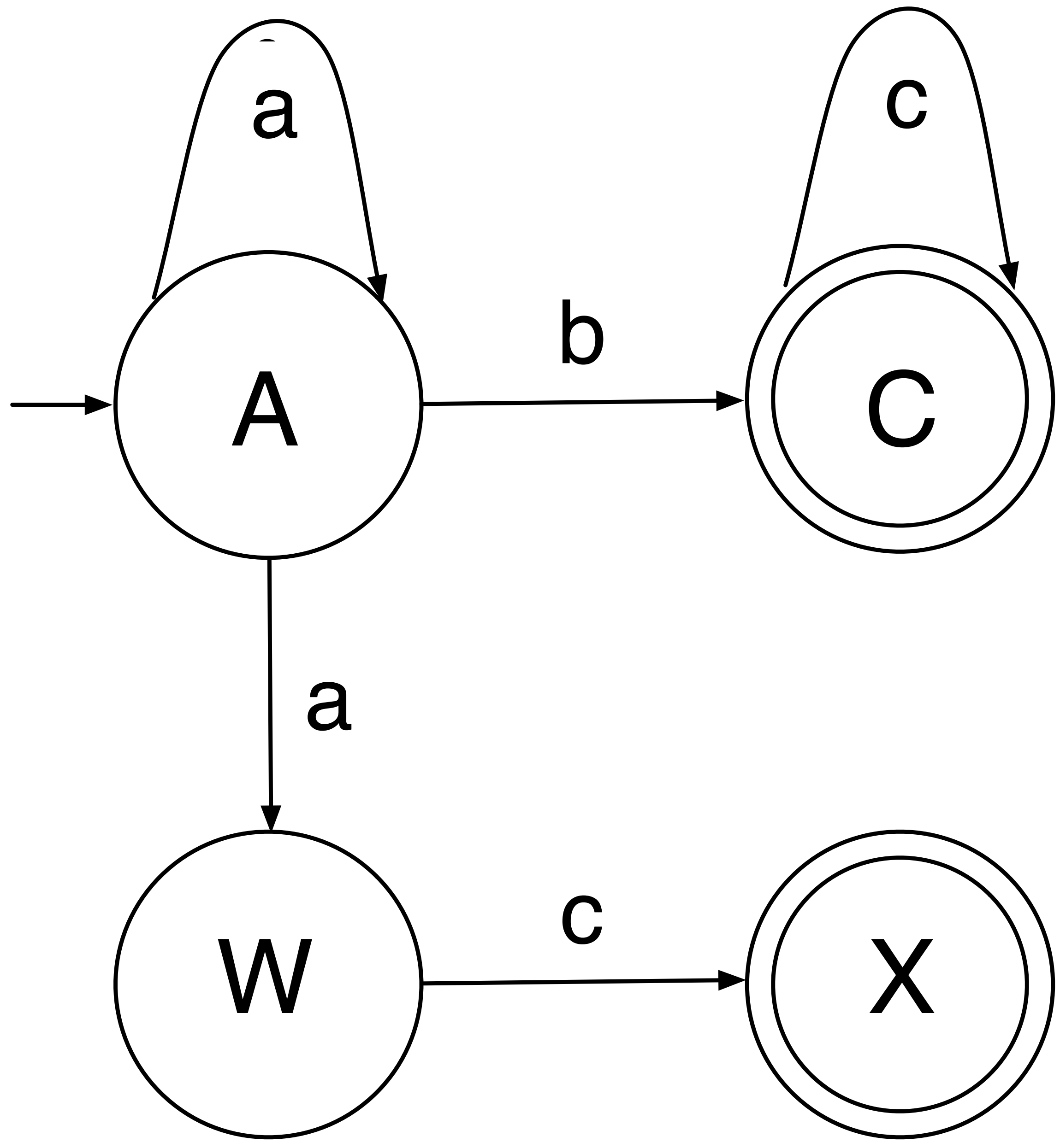
$W \rightarrow cX$

$X \rightarrow \varepsilon$

Example

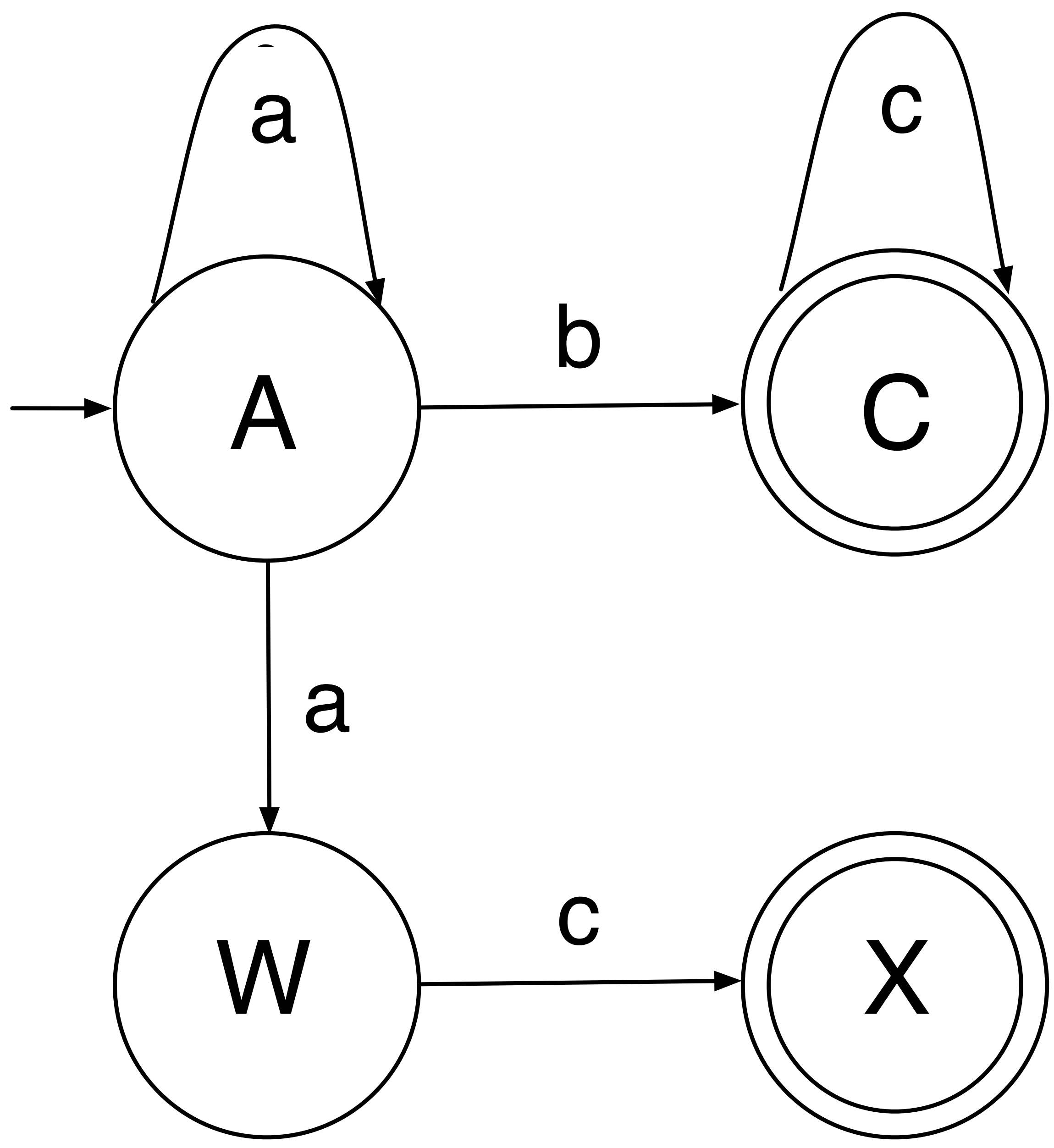


Example



$A \rightarrow aA \mid bC \mid aW$

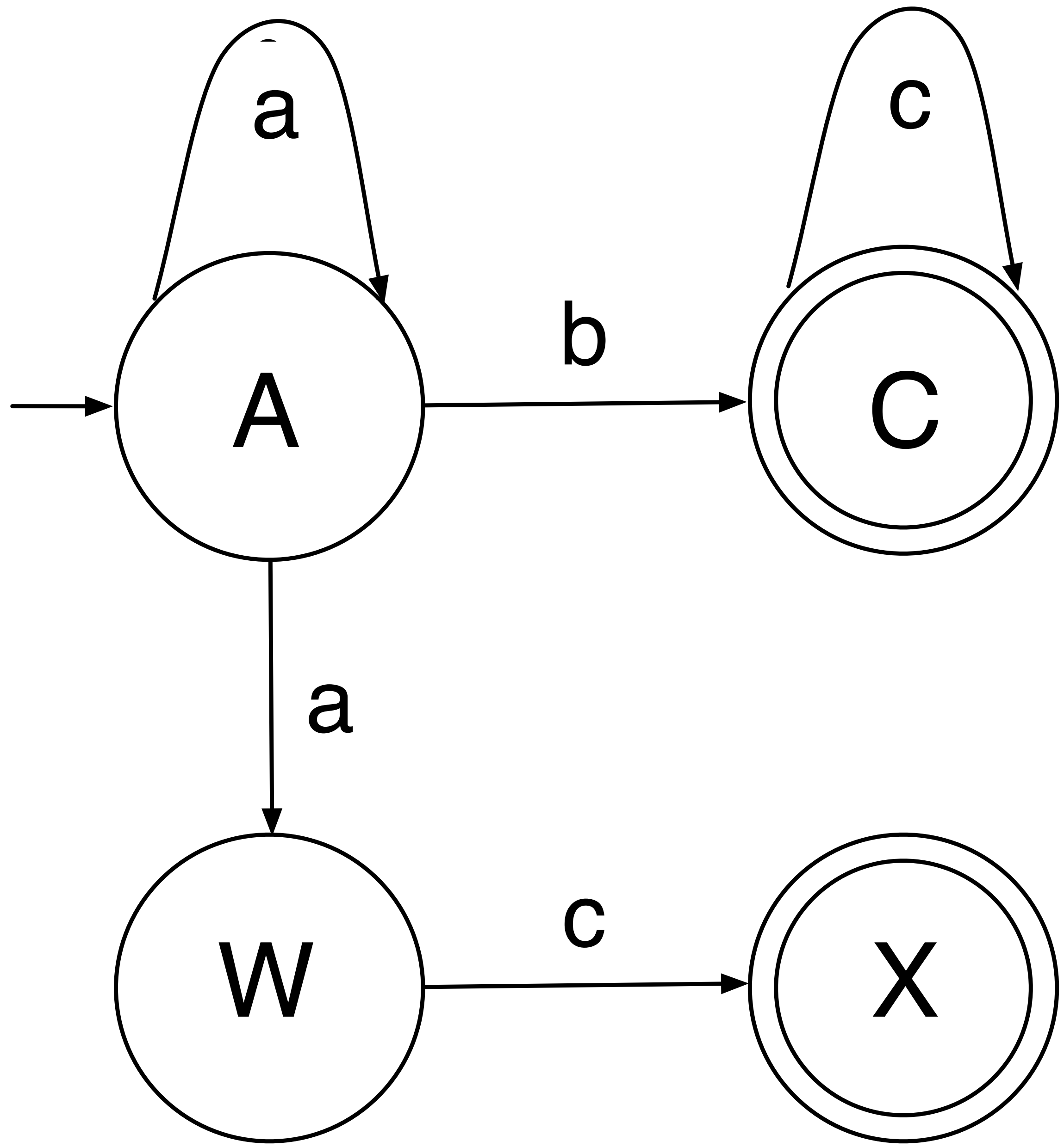
Example



$A \rightarrow aA \mid bC \mid aW$

$C \rightarrow cC \mid \varepsilon$

Example



$A \rightarrow aA \mid bC \mid aW$

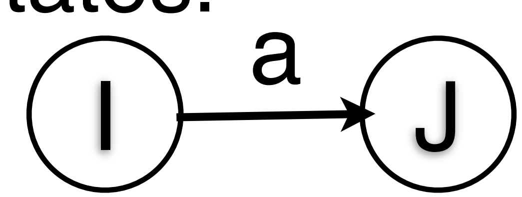
$C \rightarrow cC \mid \varepsilon$

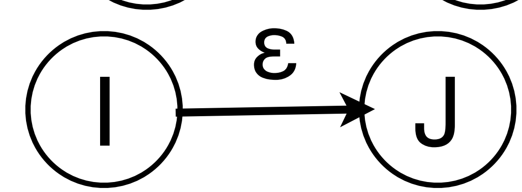
$W \rightarrow c$

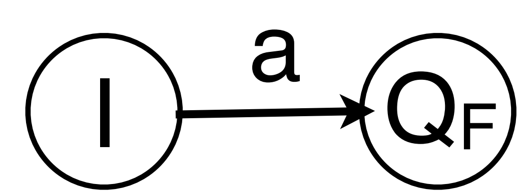
Portland State University

From Regular Grammar to FSM

Hein Algorithm 11.12

1. Transform the grammar so that all productions are of the form $A \rightarrow x$ or $A \rightarrow xB$, where x is either a single letter or ε.
2. The start state of the NFA is the grammar's start symbol.
3. Create state Q_F and add it to the set F of final states.
4. For each production $I \rightarrow aJ$, create the transition 

5. For each production $I \rightarrow J$, create the transition 

6. For each production $K \rightarrow \varepsilon$, add K to the set of final states F
7. For each production $I \rightarrow a$, create the transition 

Example

$S \rightarrow a$

$S \rightarrow B$

$B \rightarrow \varepsilon$

$B \rightarrow bB$

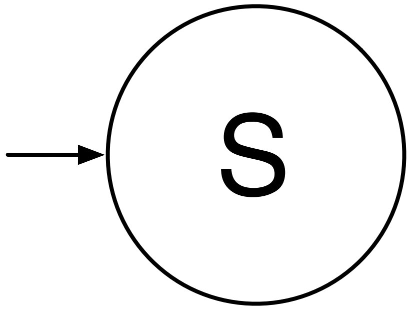
Example

$S \rightarrow a$

$S \rightarrow B$

$B \rightarrow \varepsilon$

$B \rightarrow bB$



Example

$S \rightarrow a$

$S \rightarrow B$

$B \rightarrow \varepsilon$

$B \rightarrow bB$

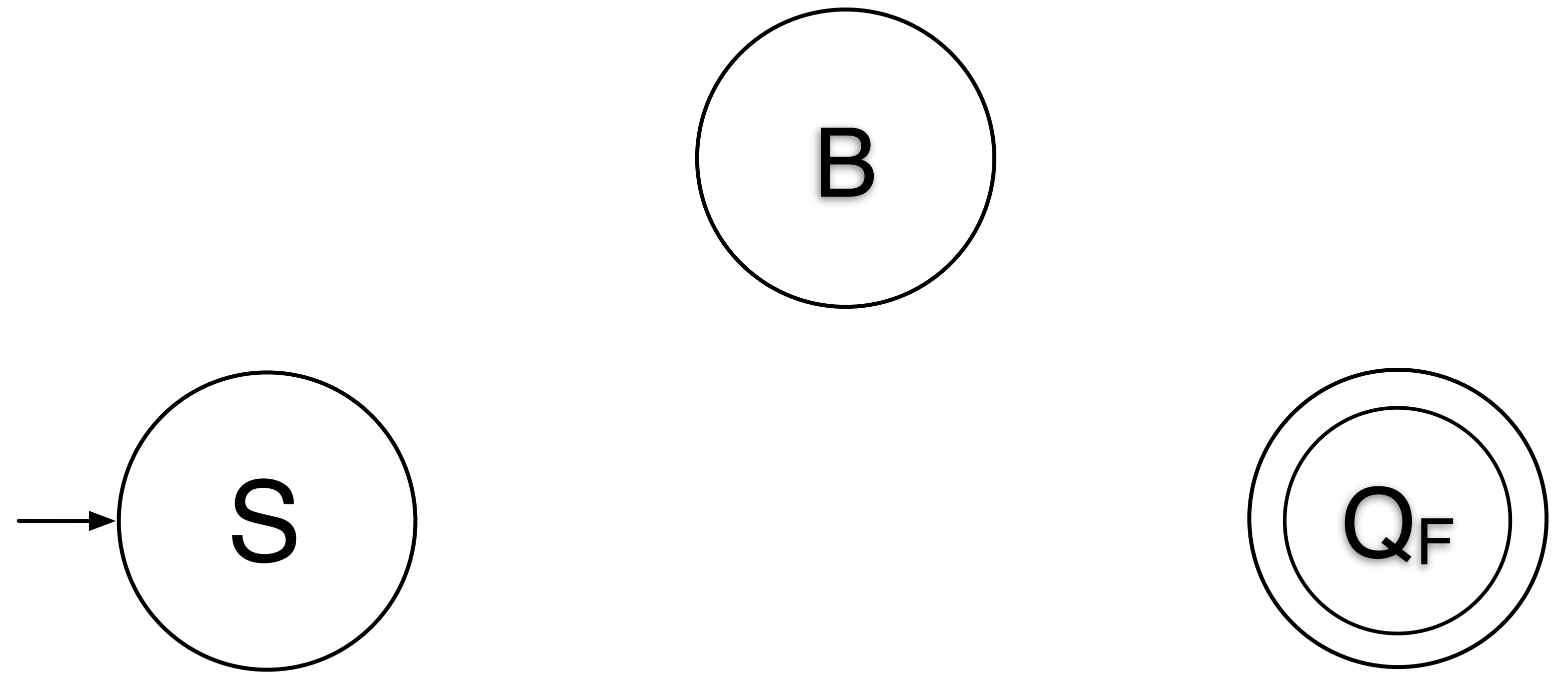
Example

S → a

S → B

B → ε

B → bB



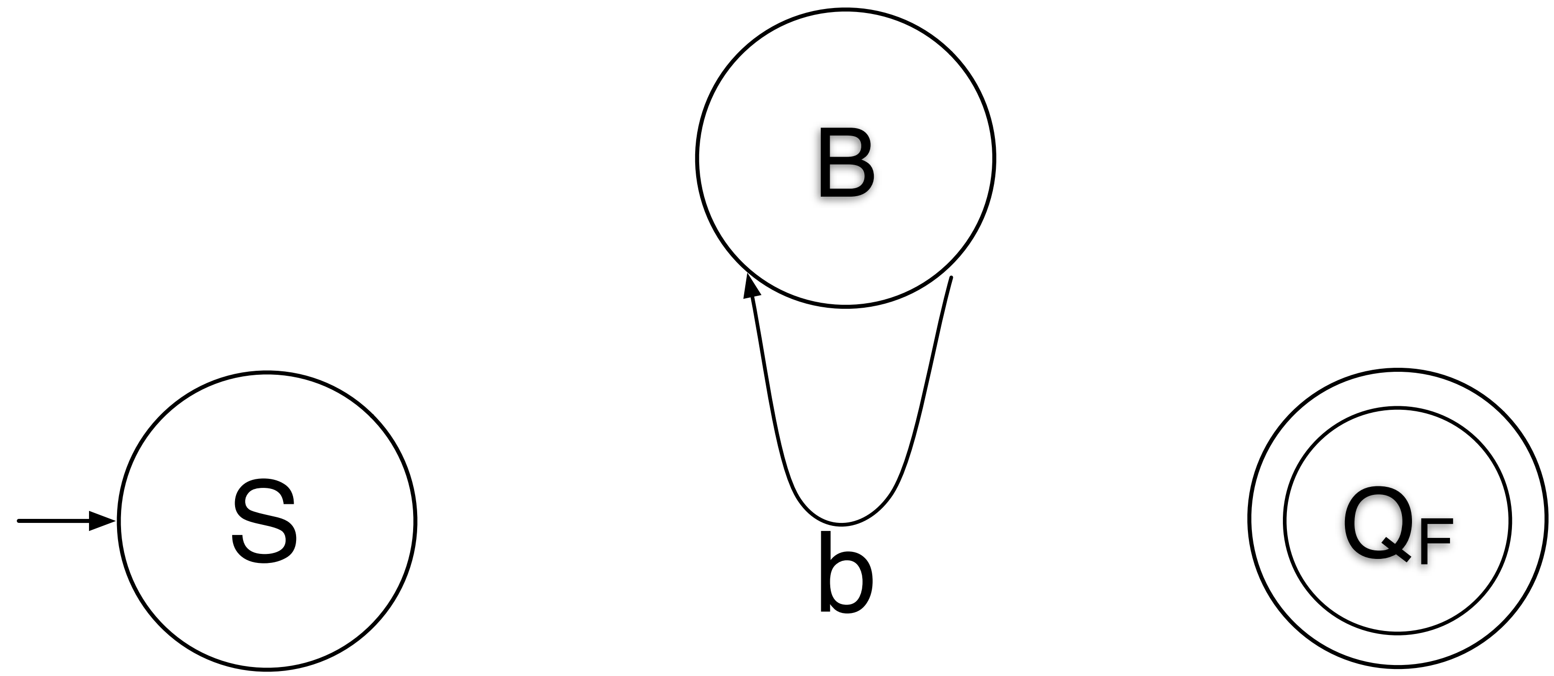
Example

S→a

S→B

B→ε

B→bB



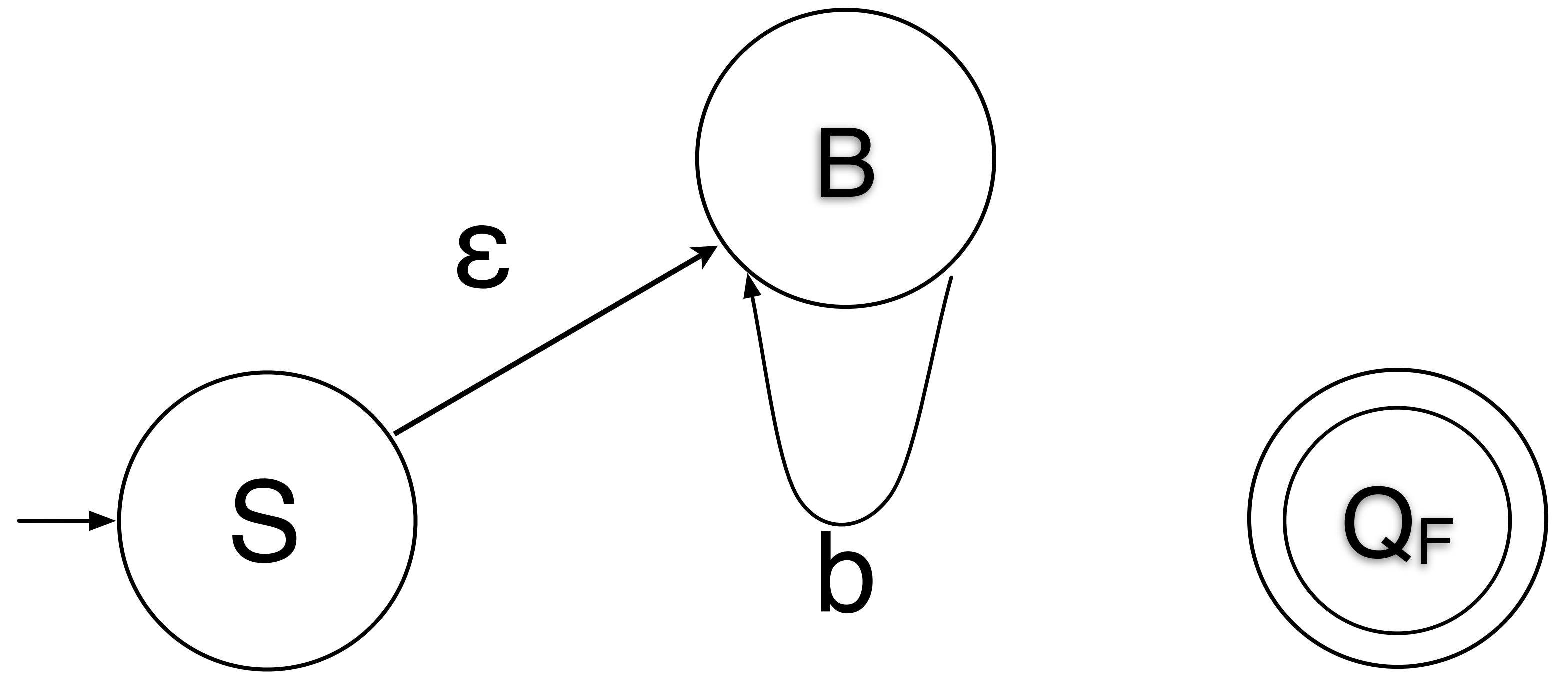
Example

$S \rightarrow a$

$S \rightarrow B$

$B \rightarrow \varepsilon$

$B \rightarrow bB$



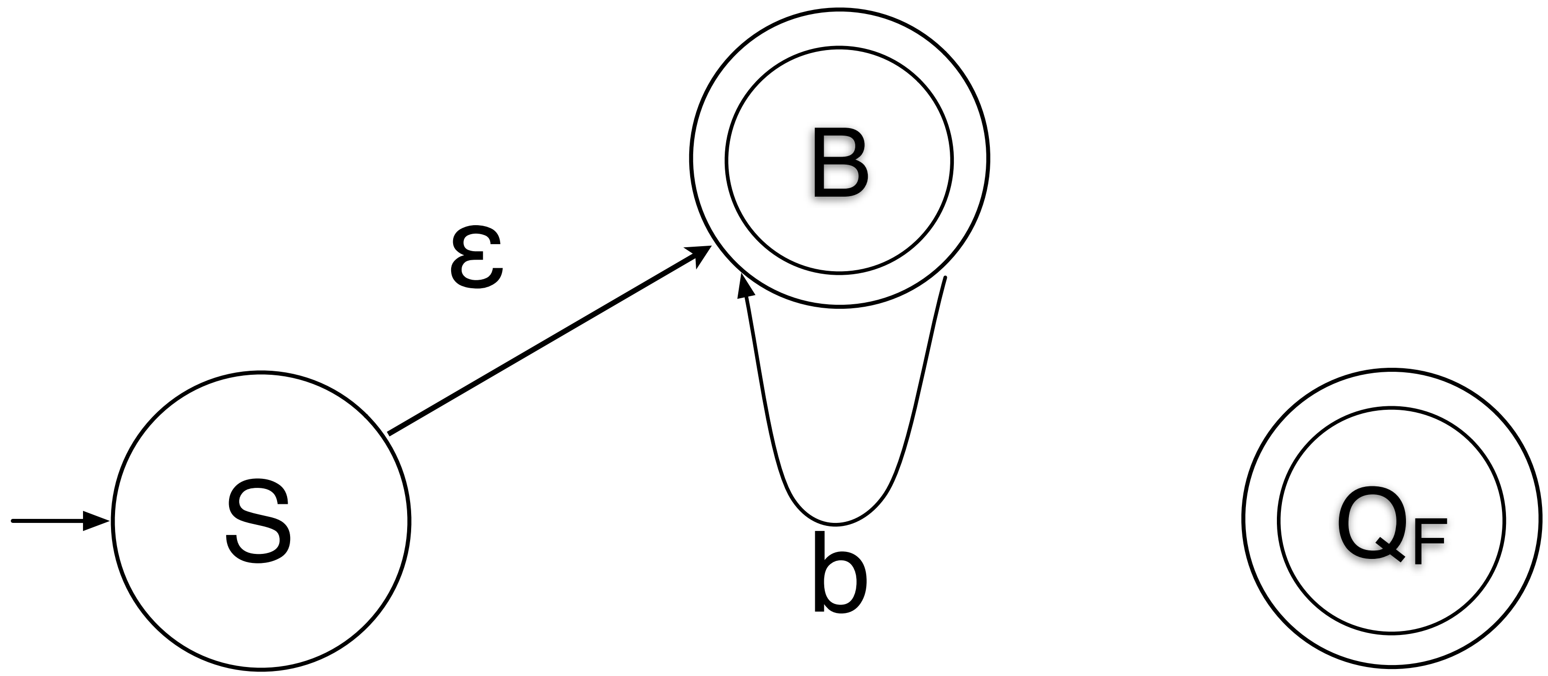
Example

$S \rightarrow a$

$S \rightarrow B$

$B \rightarrow \varepsilon$

$B \rightarrow bB$



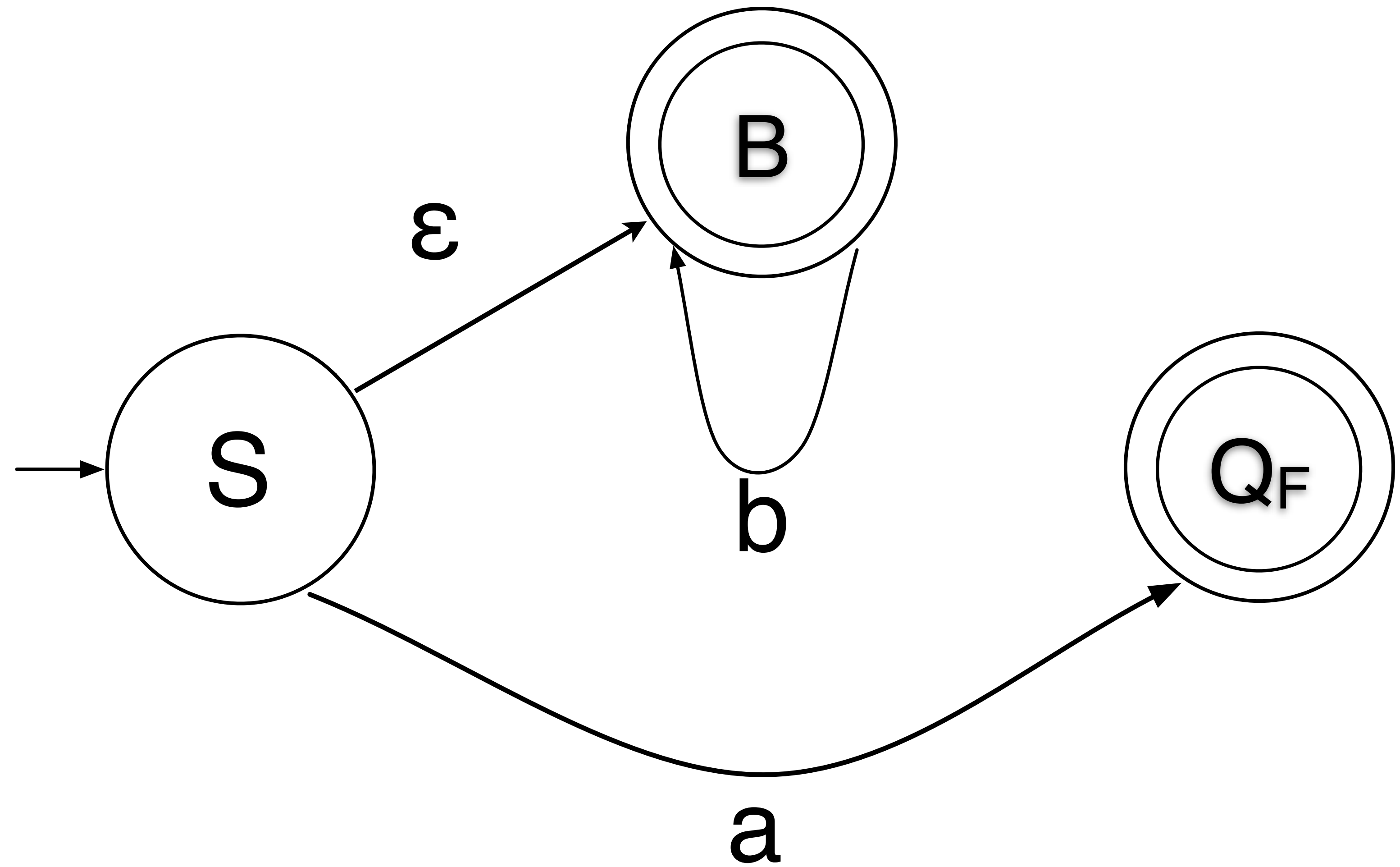
Example

$S \rightarrow a$

$S \rightarrow B$

$B \rightarrow \varepsilon$

$B \rightarrow bB$



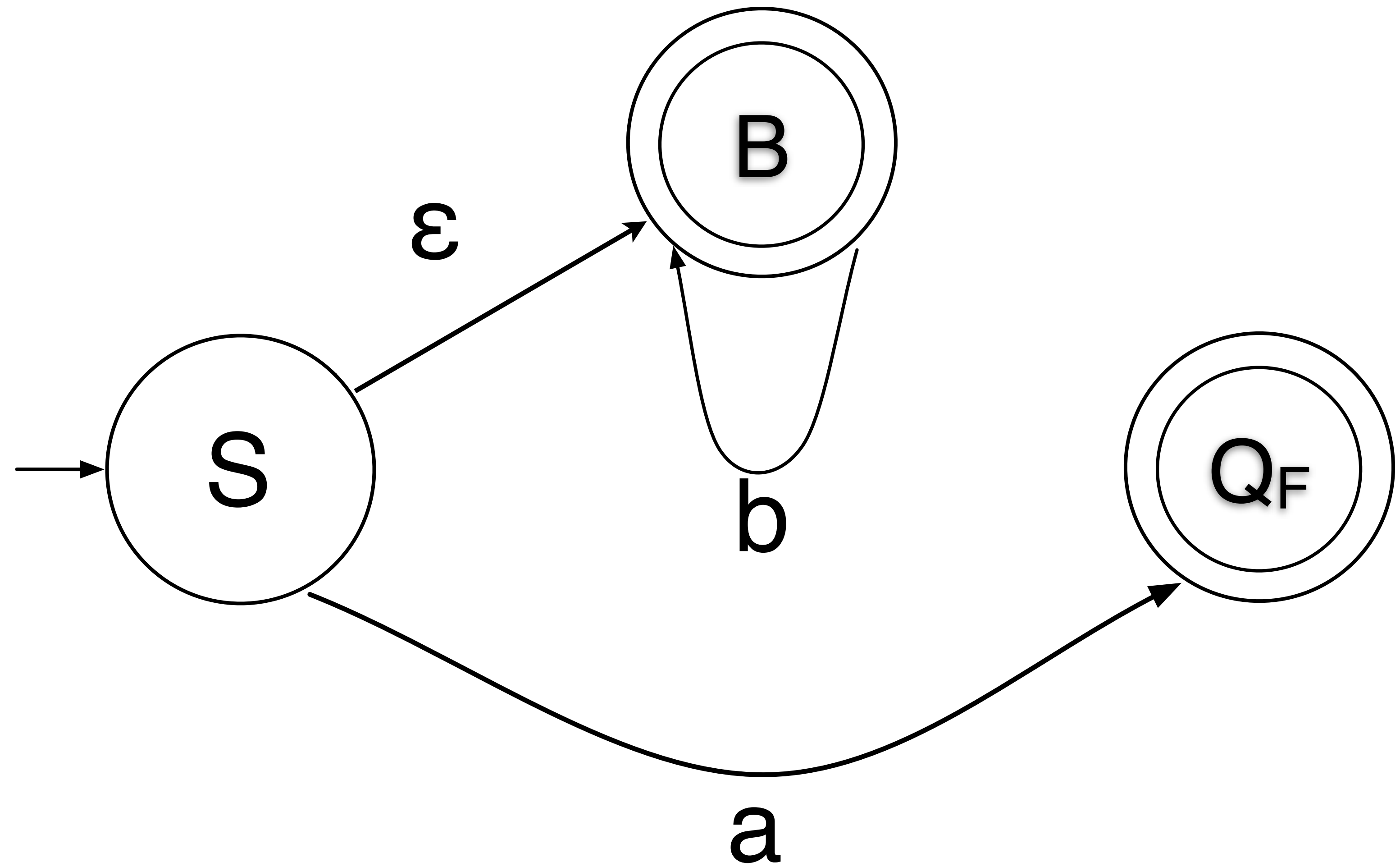
Example

$S \rightarrow a$

$S \rightarrow B$

$B \rightarrow \varepsilon$

$B \rightarrow bB$



What's the language?

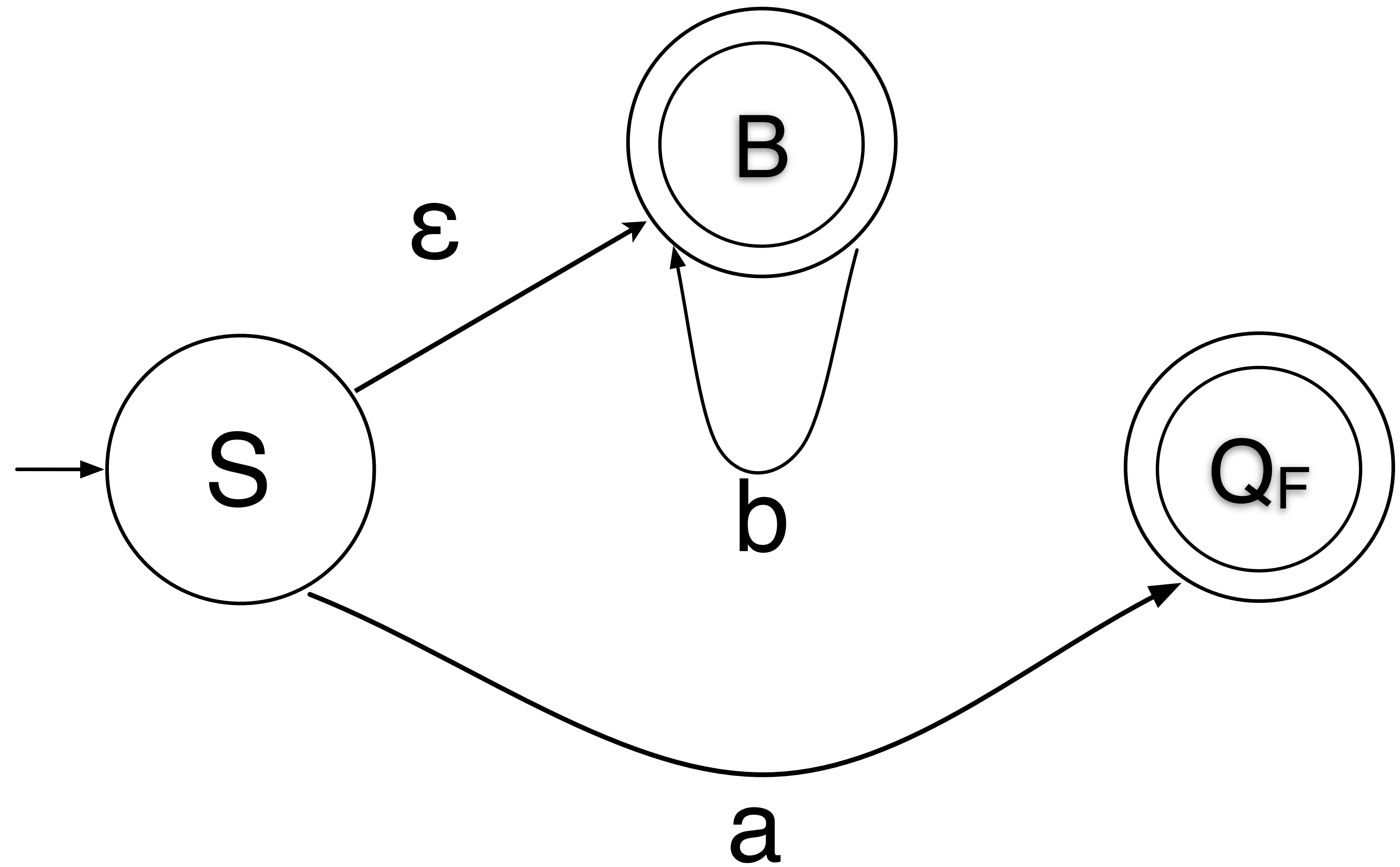
Example

$S \rightarrow a$

$S \rightarrow B$

$B \rightarrow \varepsilon$

$B \rightarrow bB$



What's the language? a + b*

Language Indistinguishability

- Consider a language L over an alphabet A.
- Two strings $x, y \in A^*$ are L-indistinguishable if for all $z \in A^*$, $xz \in L$ whenever $yz \in L$. We write $x \equiv_L y$
- $\equiv_L$ is an equivalence relation
- The *index* of L is the number of equivalence classes induced by $\equiv_L$

Portland State University

Example: $L = a + b^*$

- $a + b^* = \{ \varepsilon, a, b, bb, bbb, bbbb, \ldots \}$

  $a \equiv_L b$?

  $\varepsilon \equiv_L b$?

  $aa \equiv_L ab$?

  $ab \equiv_L bb$?

Example: $L = a + b^*$

- $a + b^* = \{ \varepsilon, a, b, bb, bbb, bbbb, \ldots\}$

  $a \equiv_L b$?

  $\varepsilon \equiv_L b$?

  $aa \equiv_L ab$?

  $ab \equiv_L bb$?

- What are the equivalence classes of $\equiv_L$?
  1. $\{a\}$
  2. $\{b, bb, bbb, bbbb, \ldots\}$
  3. $\{\varepsilon\}$
  4. everything else

Myhill-Nerode Theorem

- The equivalence relation $\equiv_L$ characterizes exactly what the state of an automaton that accepts L needs to remember about the read portion of the input:
  - if the read portion of the input is x, then the state needs to remember the equivalence class [x].
  - This is sufficient, because if $x \equiv_L y$, then it does not matter if the read portion of the input was x or y; all that matters (for deciding whether to accept or reject) is the future portion of the input, say z, because $xz \in L$ iff $yz \in L$.
  - It is also necessary, because if $x \not\equiv_L y$, then there is some possible future portion z of the input such that xz needs to be accepted and yz rejected (or vice versa).

Theorem Statement (Part A)

If the index of a language A is k, then there is a k-state DFA M_A such that $L(M_A) = A$

Mimimum-state DFA

- For any language L, there is a *unique* mimimum-state DFA that recognizes L
  - unique means "unique up to an isomorphism", that is, a renaming of the states.
- Any DFA can be transformed into a minimum-state DFA

Equivalent States

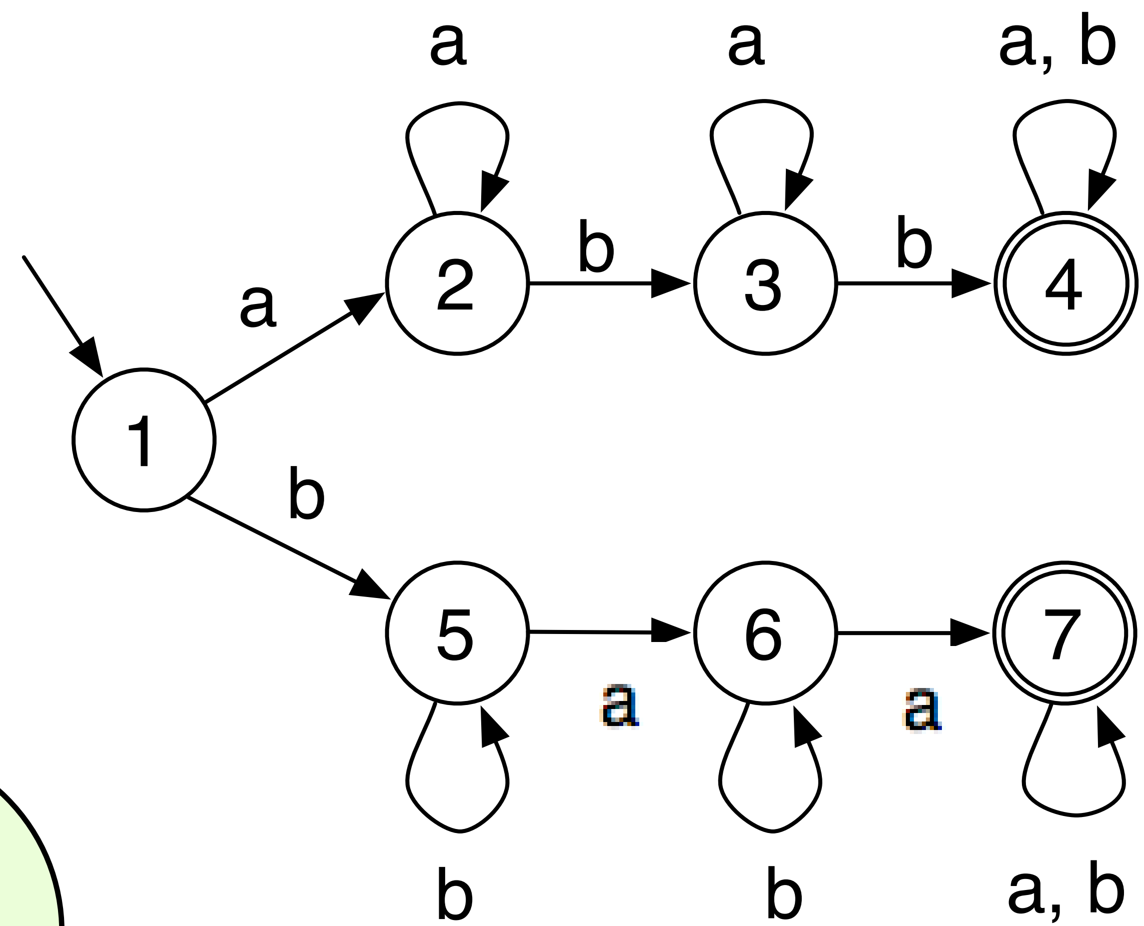
- Two states p and q in a DFA m={Q, Σ, q_0, δ, F} are *equivalent* if, for all $z \in$ Σ*, $\hat{\delta}(p, z)$ is a final state exactly when $\hat{\delta}(q, z)$ is a final state, i.e.,

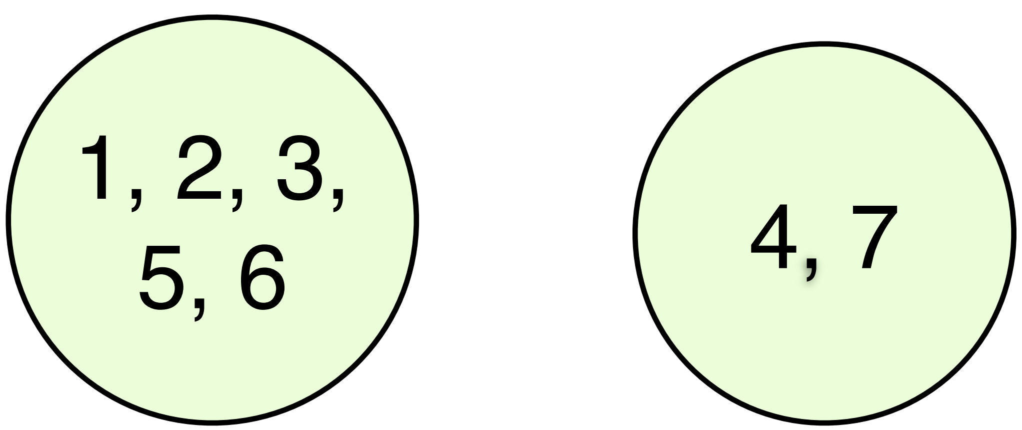
$$p \equiv q \text{ iff } \forall z \in \Sigma^*.(\hat{\delta}(p,z) \in F) \equiv (\hat{\delta}(q,z) \in F)$$

- Is this an equivalence relation?

How to Calculate State Equivalence

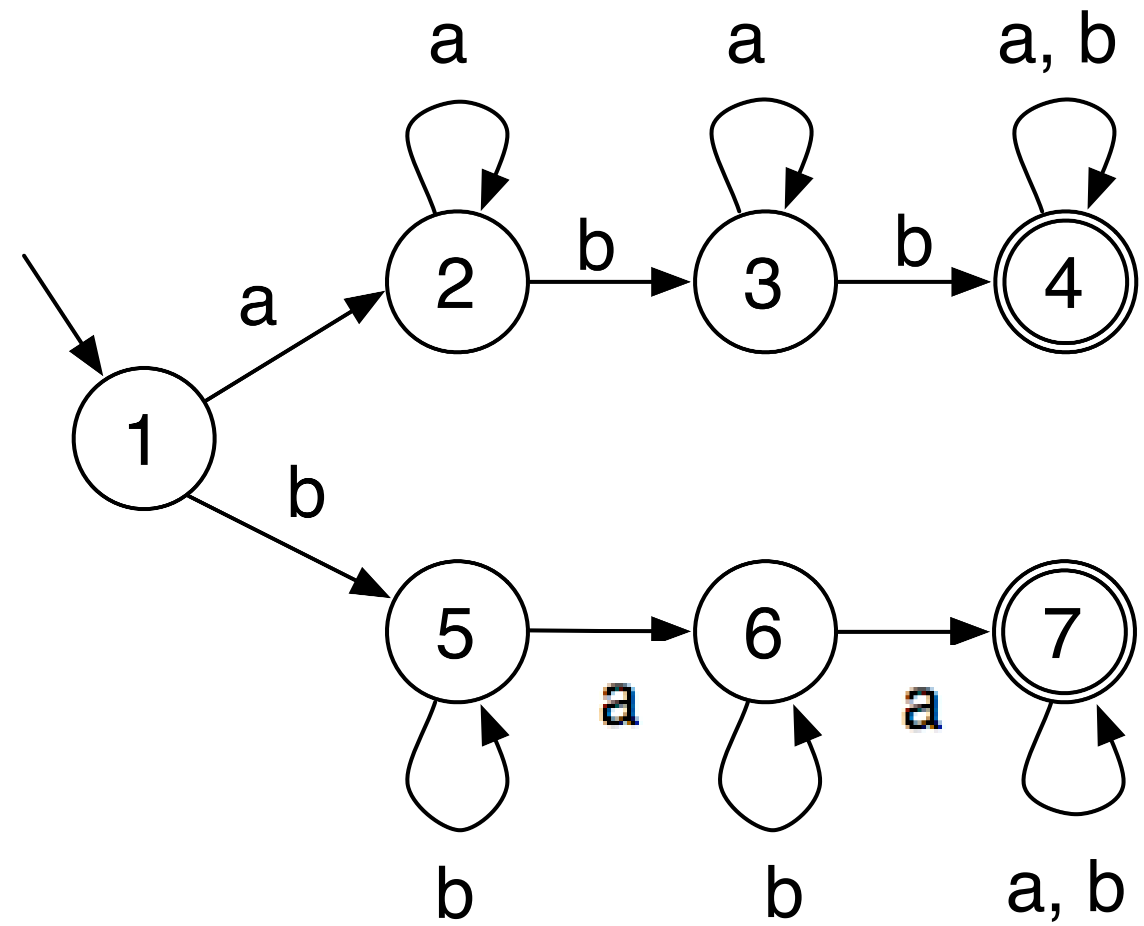
- Example:
  - 3 and 4 are not equivalent (why)?
  - First guess:





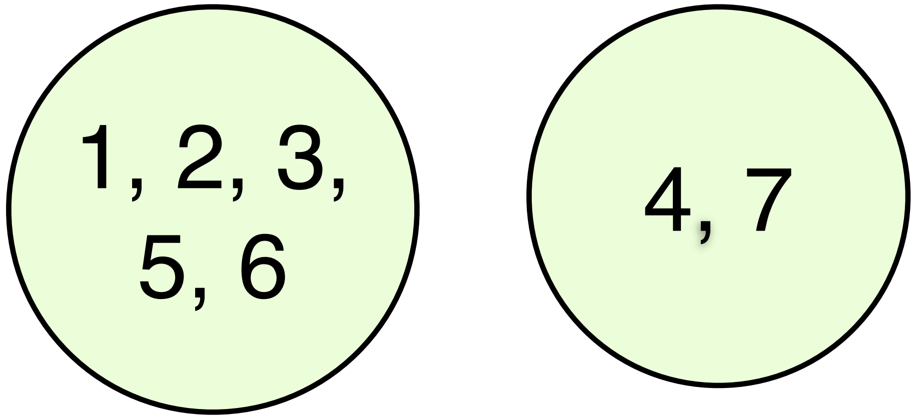
- This works for strings w of length 0

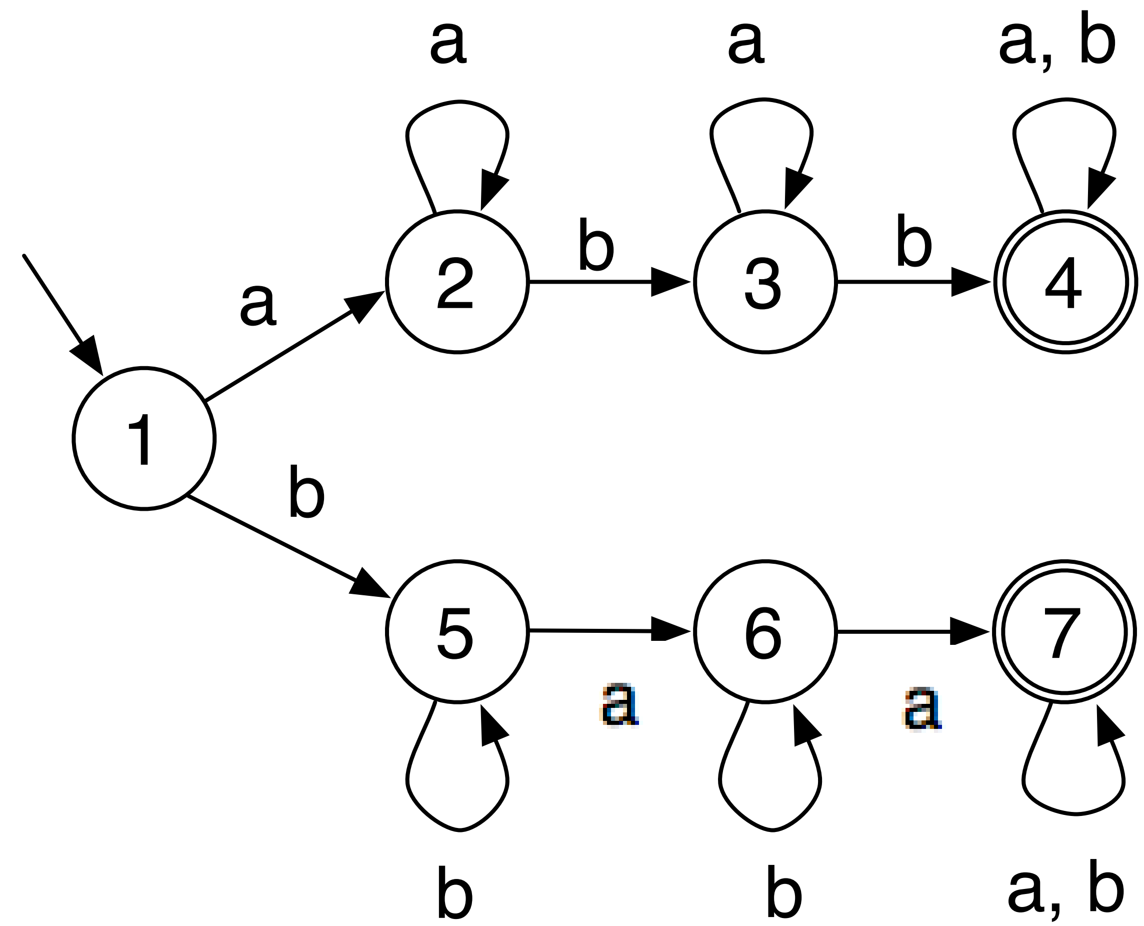
E_0: 1, 2, 3, 5, 6 | 4, 7



- Take states 1 and 2
  - for all single-character inputs, do we end up in equivalent states?
  - $\delta(1, a) = 2$; $\delta(2, a) = 2$
    $\delta(1, b) = 5$; $\delta(2, b) = 3$. $5 \equiv 3$ in E_0
- So the pair ‹1, 2› stays in the same equivalence class in the next guess, E_1

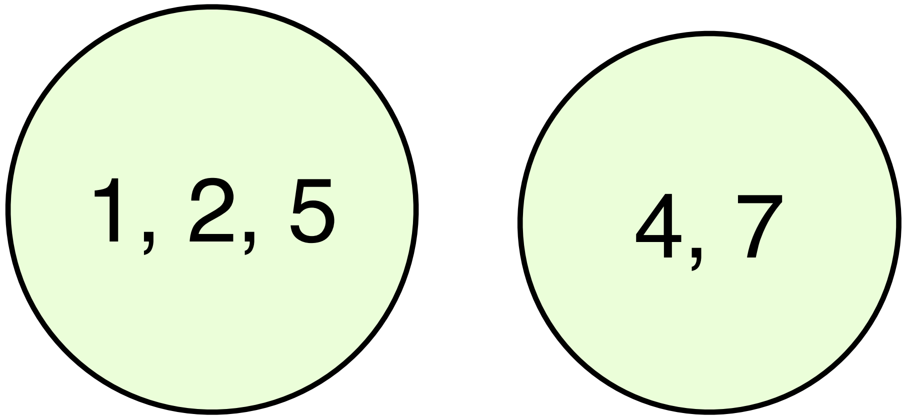
E_0:

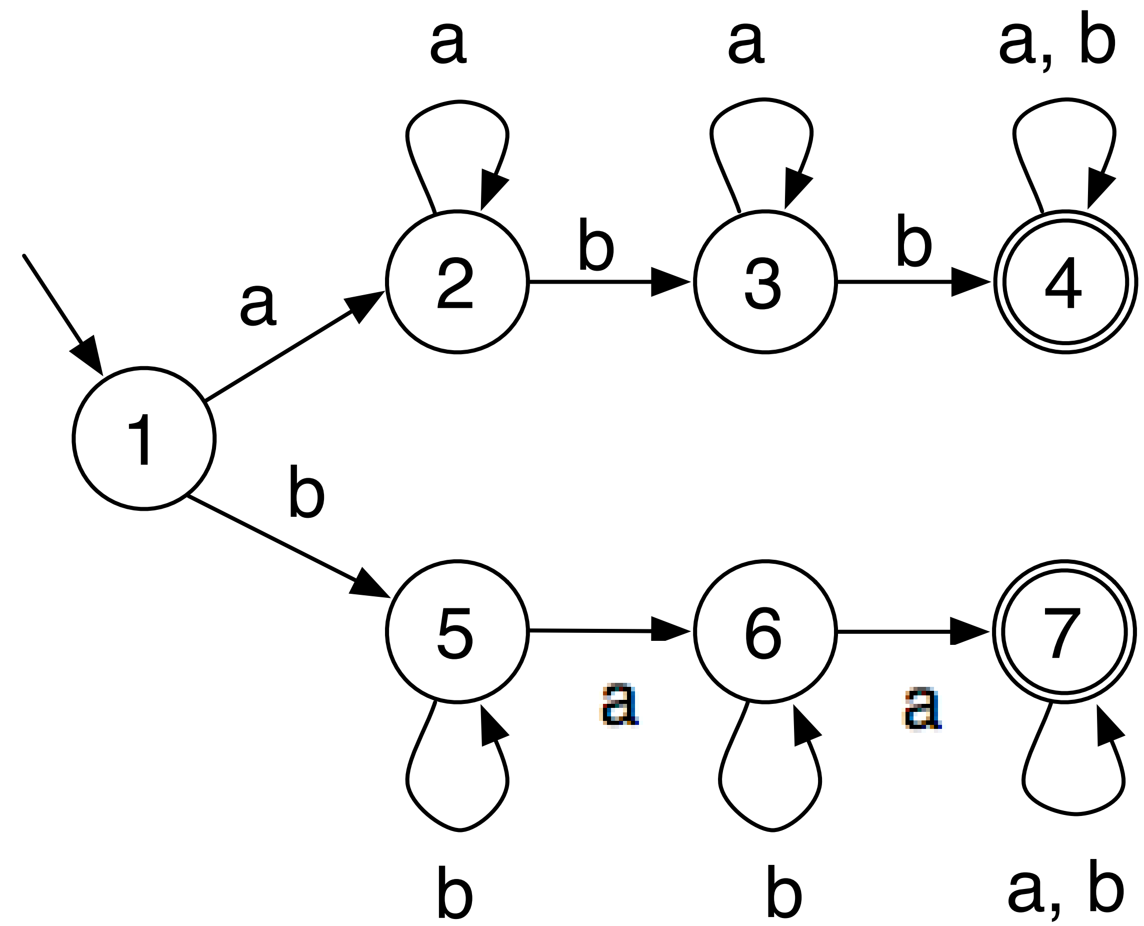




- What about states 3 and 5 ?
  - for all single-character inputs, do we end up in equivalent states?
  - $\delta(3, a) = 3$; $\delta(5, a) = 6$
    $\delta(3, b) = 4$; $\delta(5, b) = 5$. $5 \not\equiv 4$ in E_0
- So the states 3 and 5 are *not* in the same equivalence class in the next guess, E_1

E_1:





- What about states 1 and 5 ?
  - for all single-character inputs, do we end up in equivalent states?
  - $\delta(1, a) = 2$; $\delta(5, a) = 6$
    $\delta(1, b) = 5$; $\delta(5, b) = 5$. $2 \not\equiv 6$ in E_1
- So the states 1 and 5 are *not* in the same equivalence class in the next guess, E_2

- Then we repeat to get the next guess E_2
- The equivalence relation E_n represents states that act in the same way after reading input strings of length n
- Remember, a relation is nothing more than a set of pairs.
- So we build
  $E_0 \supseteq E_1 \supseteq E_2 \supseteq \ldots$
- When do we stop?

Minimizing a DFA

How can we easily compute whether or not two states p and q in a DFA are equivalent?

- Suppose that they are not equivalent:

Then some (finite) string z will be accepted when the machine starts in p, and rejected when the machine starts in q.

$$p \not\equiv q \text{ iff } \exists z \in \Sigma^*.(\hat{\delta}(p,z) \in F) \not\equiv (\hat{\delta}(q,z) \in F)$$

Portland State UNIVERSITY

Minimizing a DFA

How can we easily compute whether or not two states p and q in a DFA are equivalent?

- Suppose that they are not equivalent:

Then some (finite) string z will be accepted when the machine starts in p, and rejected when the machine starts in q.

$$p \not\equiv q \text{ iff } \exists z \in \Sigma^*.(\hat{\delta}(p,z) \in F) \not\equiv (\hat{\delta}(q,z) \in F)$$

$$p \equiv q \text{ iff } \forall z \in \Sigma^*.(\hat{\delta}(p,z) \in F) \equiv (\hat{\delta}(q,z) \in F)$$

Computing sets of equivalent states

$$E_0 \ni \langle p, q \rangle \text{ where } p \in F \equiv q \in F$$

$$E_1 = E_0 \setminus \{\langle p, q \rangle \mid \exists a \in \Sigma . \langle \delta(p, a), \delta(q, a) \rangle \notin E_0\}$$

$$\vdots$$

$$E_{n+1} = E_n \setminus \{\langle p, q \rangle \mid \exists\, a \in \Sigma . \langle \delta(p, a), \delta(q, a) \rangle \notin E_n\}$$

Constructing a Minimal DFA

Hein Construction 11.10

Algorithm to Construct a Minimum-State DFA (11.10)

Given: A DFA with set of states S and transition table T. Assume that all states that cannot be reached from the start state have already been thrown away.

Output: A minimum-state DFA recognizing the same regular language as the input DFA.

1. Construct the equivalent pairs of states by calculating the descending sequence of sets of pairs $E_0 \supset E_1 \supset \cdots$ defined as follows:

   $E_0 = \{\{s, t\} \mid s$ and t are distinct and either both states are final or both states are nonfinal$\}$.

   $E_{i+1} = \{\{s, t\} \mid \{s, t\} \in E_i$ and for every $a \in A$ either $T(s, a) = T(t, a)$ or or $\{T(s, a), T(t, a)\} \in E_i\}$.

   The computation stops when $E_k = E_{k+1}$ for some index k. E_k is the desired set of equivalent pairs.

2. Use the equivalence relation generated by the pairs in E_k to partition S into a set of equivalence classes. These equivalence classes are the states of the new DFA.
3. The *start state* is the equivalence class containing the start state of the input DFA.
4. A *final state* is any equivalence class containing a final state of the input DFA.
5. The transition table T_{min} for the minimum-state DFA is defined as follows, where $[s]$ denotes the equivalence class containing s and a is any letter: $T_{\text{min}}([s], a) = [T(s, a)]$.