Formal Languages, Automata and Codes

Oleg Gutik

Lecture 4

The first type of automaton we study in detail are finite accepters that are deterministic in their operation. We start with a precise formal definition of deterministic accepters.

The following Figures represent two simple automata, with some common features:

2.1 Deterministic Finite Accepters: Deterministic Accepters and Transition Graphs

The first type of automaton we study in detail are finite accepters that are deterministic in their operation. We start with a precise formal definition of deterministic accepters.

The following Figures represent two simple automata, with some common features:

The first type of automaton we study in detail are finite accepters that are deterministic in their operation. We start with a precise formal definition of deterministic accepters.

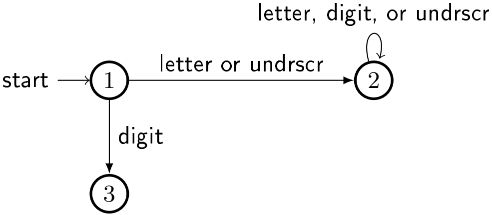
The following Figures represent two simple automata, with some common features:

The first type of automaton we study in detail are finite accepters that are deterministic in their operation. We start with a precise formal definition of deterministic accepters.

The following Figures represent two simple automata, with some common features:

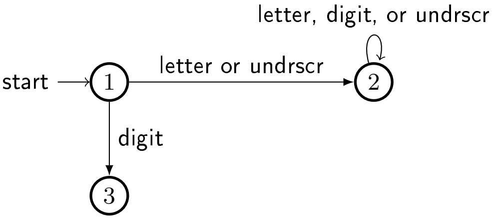
The first type of automaton we study in detail are finite accepters that are deterministic in their operation. We start with a precise formal definition of deterministic accepters.

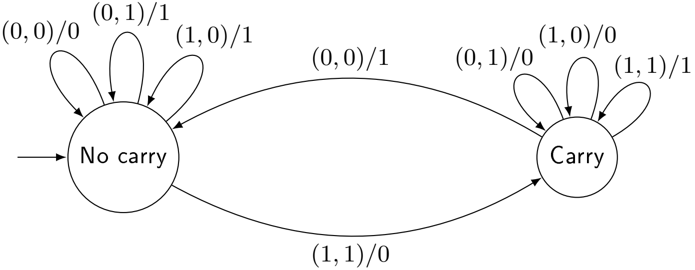
The following Figures represent two simple automata, with some common features:



The first type of automaton we study in detail are finite accepters that are deterministic in their operation. We start with a precise formal definition of deterministic accepters.

The following Figures represent two simple automata, with some common features:

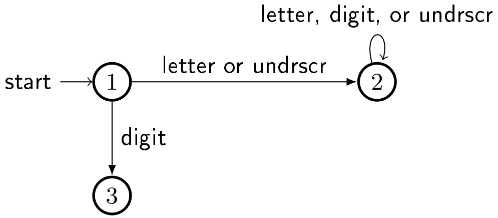


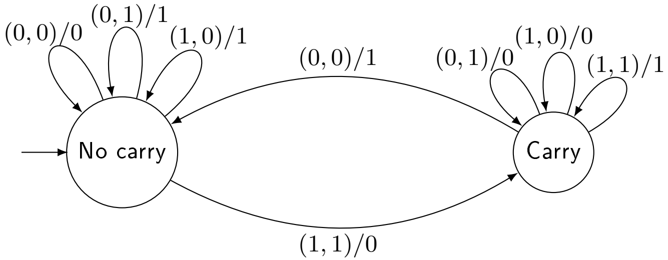


The following Figures represent two simple automata, with some common features:

- Both have a finite number of internal states.
- Both process an input string, consisting of a sequence of symbols.
- Both make transitions from one state to another, depending on the current state and the current input symbol.

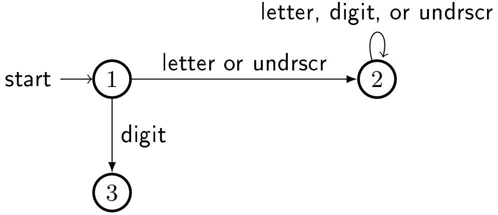
The following Figures represent two simple automata, with some common features:

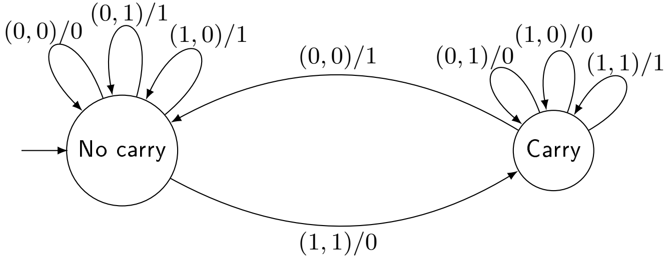




- Both have a finite number of internal states.
- Both process an input string, consisting of a sequence of symbols.
- Both make transitions from one state to another, depending on the current state and the current input symbol.

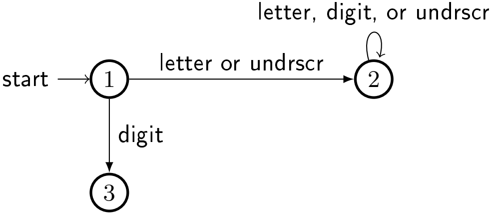
The following Figures represent two simple automata, with some common features:

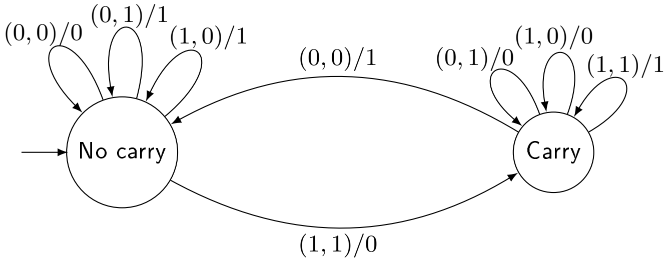




- Both have a finite number of internal states.
- Both process an input string, consisting of a sequence of symbols.
- Both make transitions from one state to another, depending on the current state and the current input symbol.

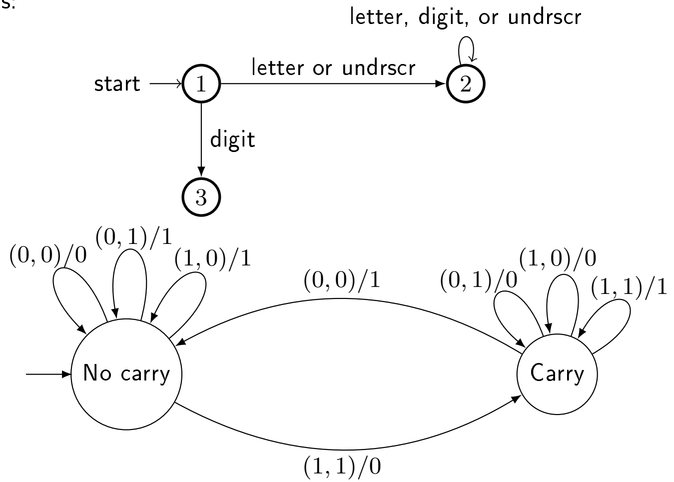
The following Figures represent two simple automata, with some common features:





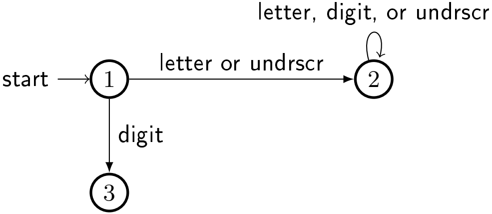
- Both have a finite number of internal states.
- Both process an input string, consisting of a sequence of symbols.
- Both make transitions from one state to another, depending on the current state and the current input symbol.

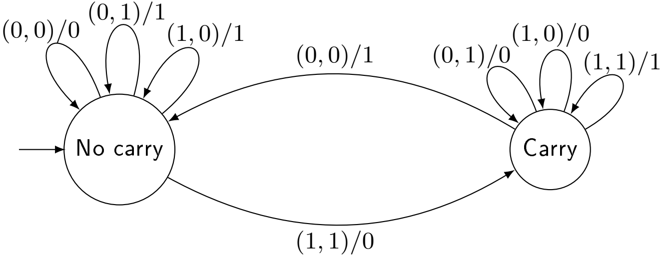
The following Figures represent two simple automata, with some common features:



- Both have a finite number of internal states.
- Both process an input string, consisting of a sequence of symbols.
- Both make transitions from one state to another, depending on the current state and the current input symbol.

The following Figures represent two simple automata, with some common features:



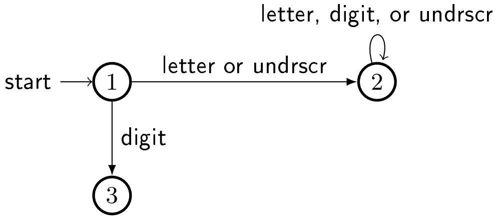


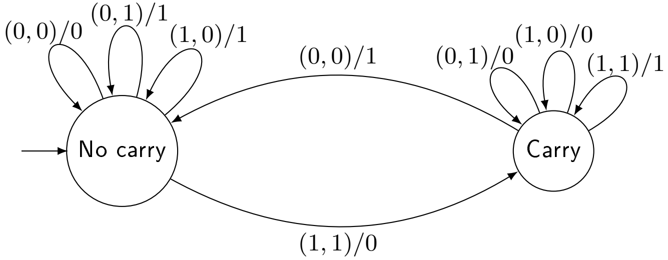
- Both have a finite number of internal states.
- Both process an input string, consisting of a sequence of symbols.
- Both make transitions from one state to another, depending on the current state and the current input symbol.

The following Figures represent two simple automata, with some common features:

- Both produce some output, but in a slightly different form. The automaton in the first Figure only accepts or rejects the input; the automaton in the second Figure generates an output string.

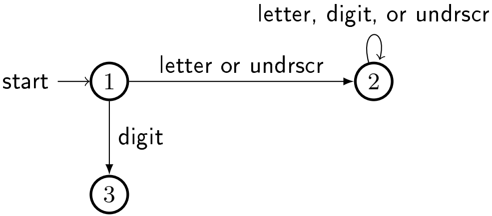
The following Figures represent two simple automata, with some common features:

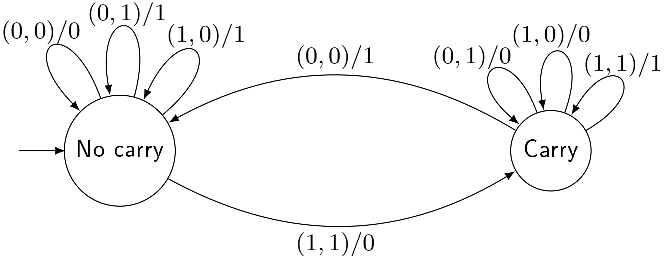




- Both produce some output, but in a slightly different form. The automaton in the first Figure only accepts or rejects the input; the automaton in the second Figure generates an output string.

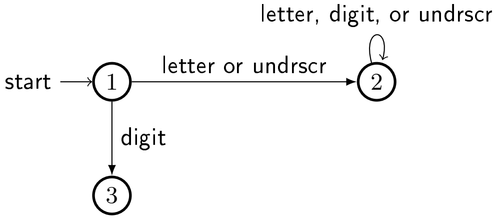
The following Figures represent two simple automata, with some common features:

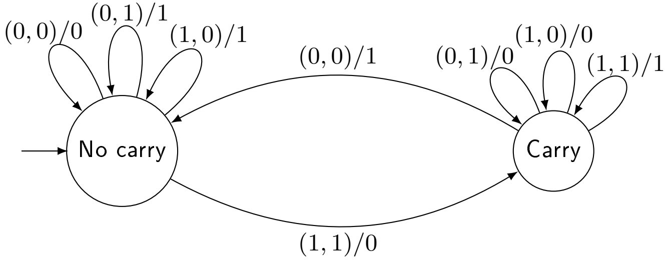




- Both produce some output, but in a slightly different form. The automaton in the first Figure only accepts or rejects the input; the automaton in the second Figure generates an output string.

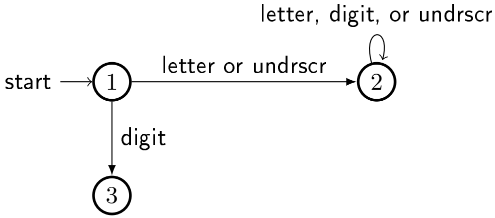
The following Figures represent two simple automata, with some common features:

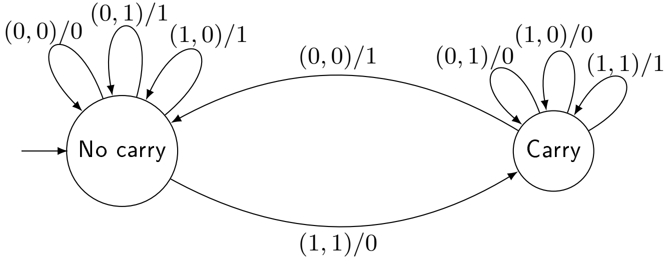




- Both produce some output, but in a slightly different form. The automaton in the first Figure only accepts or rejects the input; the automaton in the second Figure generates an output string.

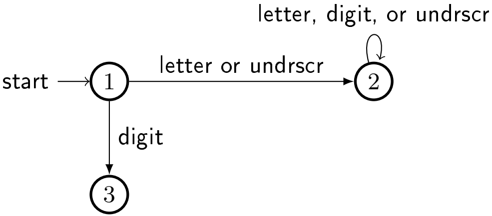
The following Figures represent two simple automata, with some common features:

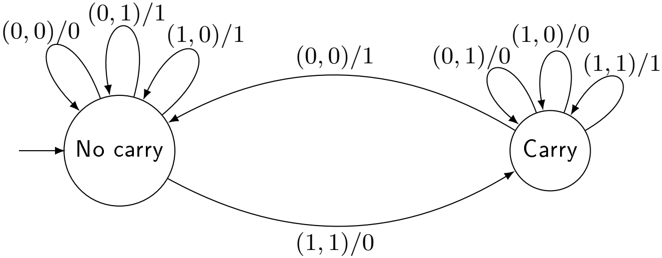




- Both produce some output, but in a slightly different form. The automaton in the first Figure only accepts or rejects the input; the automaton in the second Figure generates an output string.

The following Figures represent two simple automata, with some common features:





- Both produce some output, but in a slightly different form. The automaton in the first Figure only accepts or rejects the input; the automaton in the second Figure generates an output string.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta: Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta : Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta: Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta : Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta : Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta : Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta : Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta : Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta : Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta \colon Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta\colon Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta \colon Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta \colon Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta\colon Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

Notice also that both automata have a single well-defined transition at each step. All of these features are incorporated in the following definition.

Definition 2.1

A *deterministic finite accepter* or *DFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *internal states*,
- Σ is a finite set of symbols called the *input alphabet*,
- $\delta \colon Q \times \Sigma \to Q$ is a total function called the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a set of *final states*.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected.
The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

$$\delta(q_0, a) = q_1,$$

then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use transition graphs, in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some DFA M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled by the letter a represents the transition $\delta(q_0, a) = q_1$. The initial state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled by a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices*. It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a DFA to its transition graph representation and vice versa.

Example 2.1

The graph in the following Figure

represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

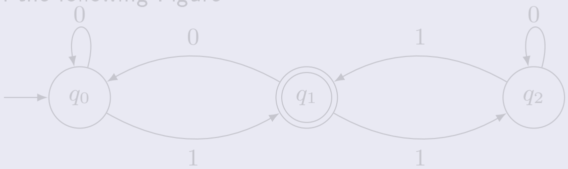
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

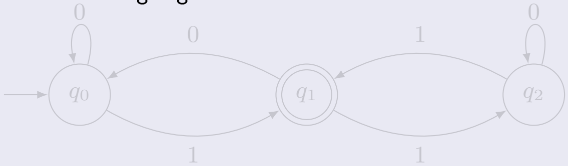
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_1\}),$$

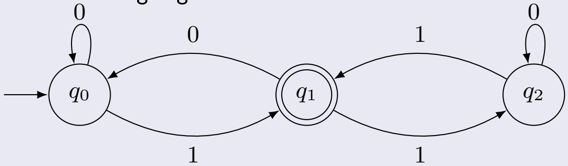
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

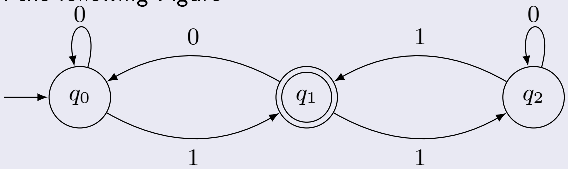
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

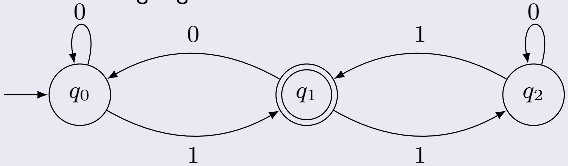
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

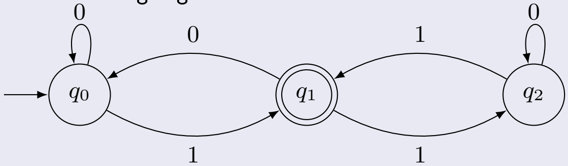
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

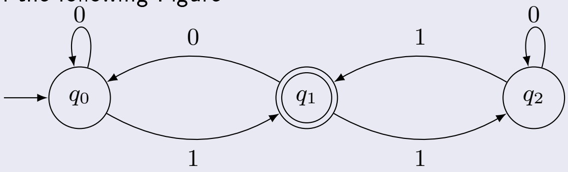
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

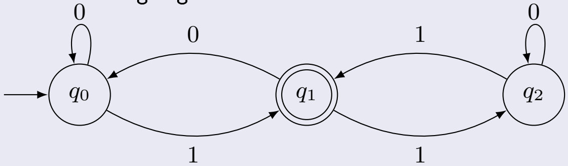
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

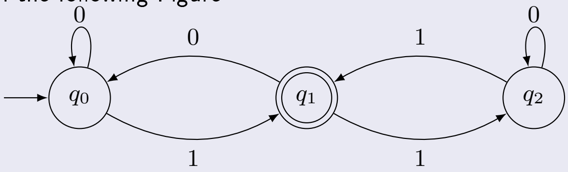
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

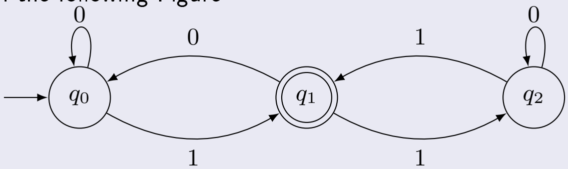
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

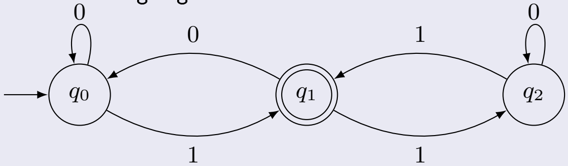
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

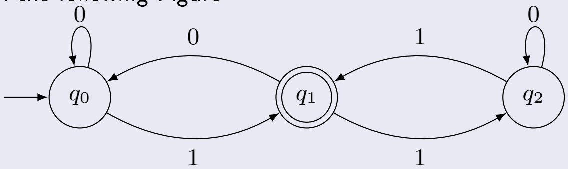
where δ is given by

$$\delta(q_0, 0) = q_0, \quad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \quad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \quad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

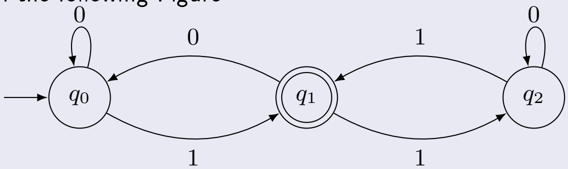
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

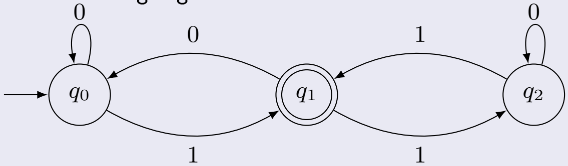
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

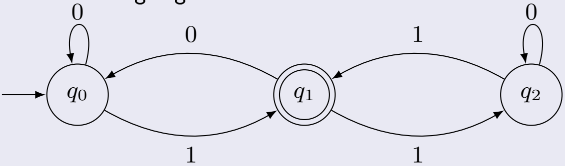
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

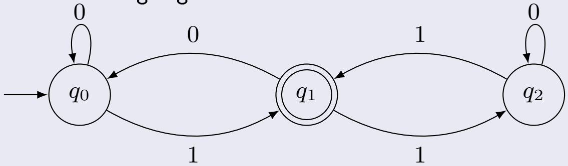
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

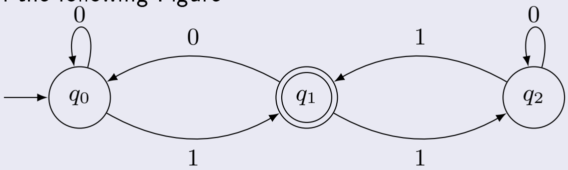
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

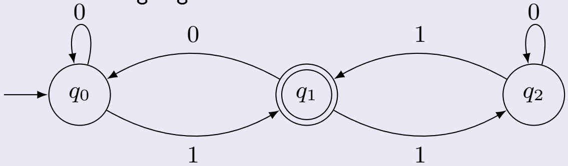
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

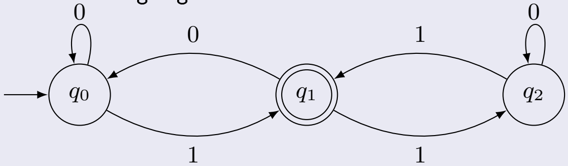
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

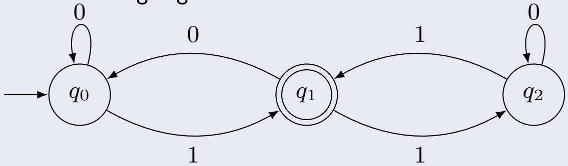
where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

Example 2.1

The graph in the following Figure



represents the DFA

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

where δ is given by

$$\delta(q_0, 0) = q_0, \qquad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_0, \qquad \delta(q_1, 1) = q_2,$$
$$\delta(q_2, 0) = q_2, \qquad \delta(q_2, 1) = q_1.$$

This DFA accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The DFA does not accept the string 00, because after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \tag{3}$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) = \\ = \delta(q_0, a) = \\ = q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \tag{3}$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) = \\ = \delta(q_0, a) = \\ = q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \quad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \quad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \quad (3)$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) = = \delta(q_0, a) = = q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \quad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \quad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \quad (3)$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) = = \delta(q_0, a) = = q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \qquad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \qquad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \qquad (3)$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) =$$

$$= \delta(q_0, a) =$$

$$= q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \qquad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \qquad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \qquad (3)$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) = \delta(q_0, a) = q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \tag{3}$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) =$$
$$= \delta(q_0, a) =$$
$$= q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \tag{3}$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) = \\ = \delta(q_0, a) = \\ = q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \qquad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \qquad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \qquad (3)$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) =$$
$$= \delta(q_0, a) =$$
$$= q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \tag{3}$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) = \\ = \delta(q_0, a) = \\ = q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \quad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \quad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \quad (3)$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) =$$

$$= \delta(q_0, a) =$$

$$= q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \tag{3}$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) =$$

$$= \delta(q_0, a) =$$

$$= q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \qquad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \qquad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \qquad (3)$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) =$$

$$= \delta(q_0, a) =$$

$$= q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \qquad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \qquad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \qquad (3)$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) = \\ = \delta(q_0, a) = \\ = q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \quad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \quad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \quad (3)$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) =$$
$$= \delta(q_0, a) =$$
$$= q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \qquad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \qquad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \qquad (3)$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) =$$
$$= \delta(q_0, a) =$$
$$= q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \quad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \quad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \quad (3)$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) =$$
$$= \delta(q_0, a) =$$
$$= q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \tag{3}$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) = \\ = \delta(q_0, a) = \\ = q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \tag{3}$$

But

$$\begin{aligned}\delta^*(q_0, a) &= \delta(\delta^*(q_0, \lambda), a) = \\ &= \delta(q_0, a) = \\ &= q_1\end{aligned}$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*: Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \tag{3}$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) = \\ = \delta(q_0, a) = \\ = q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \qquad (1)$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \qquad (2)$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \qquad (3)$$

But

$$\begin{aligned} \delta^*(q_0, a) &= \delta(\delta^*(q_0, \lambda), a) = \\ &= \delta(q_0, a) = \\ &= q_1 \end{aligned}$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

It is convenient to introduce the extended transition function $\delta^*\colon Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \tag{3}$$

But

$$\delta^*(q_0, a) = \delta(\delta^*(q_0, \lambda), a) = \\ = \delta(q_0, a) = \\ = q_1$$

Substituting this into (3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\} .$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\} .$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\} .$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\} .$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\} .$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\} .$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\} .$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\} .$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: The language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\} .$$

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A DFA will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the DFA stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\} .$$

Example 2.2

Consider the DFA in the following Figure.

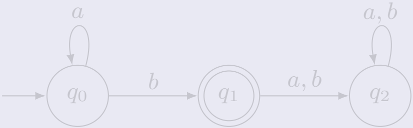
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



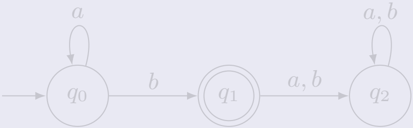
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



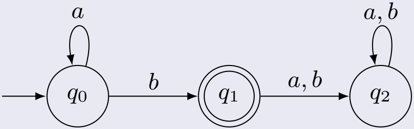
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



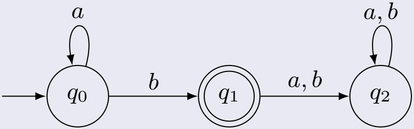
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



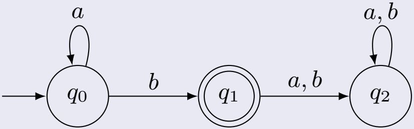
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b : n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



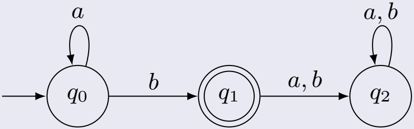
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



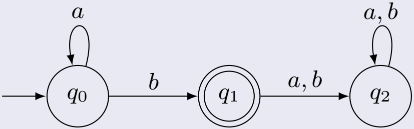
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



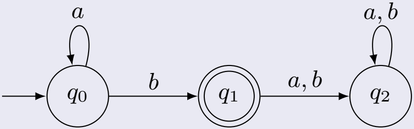
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b : n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



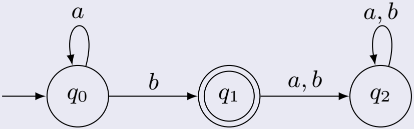
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



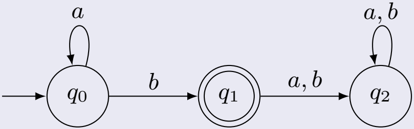
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b\colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



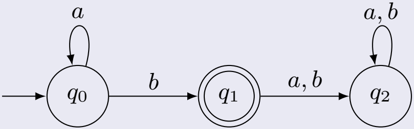
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



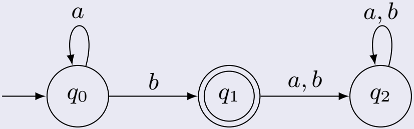
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



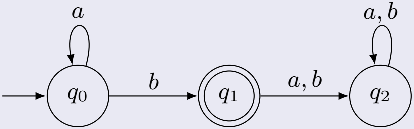
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



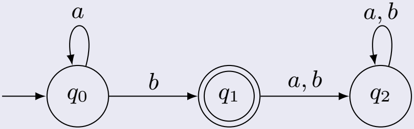
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



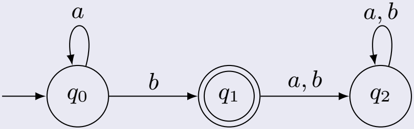
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



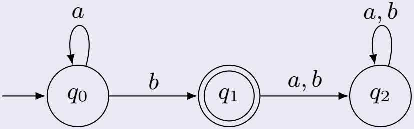
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b : n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



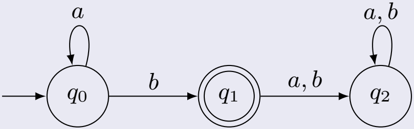
In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

Example 2.2

Consider the DFA in the following Figure.



In the drawing we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in the Figure remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the DFA goes into state q_2, from which it can never escape. The state q_2 is a trap state. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b \colon n \geqslant 0\}.$$

These examples show how convenient transition graphs are for working with finite automata. While it is possible to base all arguments strictly on the properties of the transition function and its extension through (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$
$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

the results are hard to follow. In our discussion, we use graphs, which are more intuitive, as far as possible. To do so, we must, of course, have some assurance that we are not misled by the representation and that arguments based on graphs are as valid as those that use the formal properties of δ. The following preliminary result gives us this assurance.

These examples show how convenient transition graphs are for working with finite automata. While it is possible to base all arguments strictly on the properties of the transition function and its extension through (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

the results are hard to follow. In our discussion, we use graphs, which are more intuitive, as far as possible. To do so, we must, of course, have some assurance that we are not misled by the representation and that arguments based on graphs are as valid as those that use the formal properties of δ. The following preliminary result gives us this assurance.

These examples show how convenient transition graphs are for working with finite automata. While it is possible to base all arguments strictly on the properties of the transition function and its extension through (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$
$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

the results are hard to follow. In our discussion, we use graphs, which are more intuitive, as far as possible. To do so, we must, of course, have some assurance that we are not misled by the representation and that arguments based on graphs are as valid as those that use the formal properties of δ. The following preliminary result gives us this assurance.

These examples show how convenient transition graphs are for working with finite automata. While it is possible to base all arguments strictly on the properties of the transition function and its extension through (1) and (2),

$$\delta^*(q,\lambda) = q, \tag{1}$$

$$\delta^*(q,wa) = \delta(\delta^*(q,w),a), \tag{2}$$

the results are hard to follow. In our discussion, we use graphs, which are more intuitive, as far as possible. To do so, we must, of course, have some assurance that we are not misled by the representation and that arguments based on graphs are as valid as those that use the formal properties of δ. The following preliminary result gives us this assurance.

These examples show how convenient transition graphs are for working with finite automata. While it is possible to base all arguments strictly on the properties of the transition function and its extension through (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

the results are hard to follow. In our discussion, we use graphs, which are more intuitive, as far as possible. To do so, we must, of course, have some assurance that we are not misled by the representation and that arguments based on graphs are as valid as those that use the formal properties of δ. The following preliminary result gives us this assurance.

These examples show how convenient transition graphs are for working with finite automata. While it is possible to base all arguments strictly on the properties of the transition function and its extension through (1) and (2),

$$\delta^*(q,\lambda) = q, \tag{1}$$
$$\delta^*(q,wa) = \delta(\delta^*(q,w),a), \tag{2}$$

the results are hard to follow. In our discussion, we use graphs, which are more intuitive, as far as possible. To do so, we must, of course, have some assurance that we are not misled by the representation and that arguments based on graphs are as valid as those that use the formal properties of δ. The following preliminary result gives us this assurance.

These examples show how convenient transition graphs are for working with finite automata. While it is possible to base all arguments strictly on the properties of the transition function and its extension through (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

the results are hard to follow. In our discussion, we use graphs, which are more intuitive, as far as possible. To do so, we must, of course, have some assurance that we are not misled by the representation and that arguments based on graphs are as valid as those that use the formal properties of δ. The following preliminary result gives us this assurance.

These examples show how convenient transition graphs are for working with finite automata. While it is possible to base all arguments strictly on the properties of the transition function and its extension through (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

the results are hard to follow. In our discussion, we use graphs, which are more intuitive, as far as possible. To do so, we must, of course, have some assurance that we are not misled by the representation and that arguments based on graphs are as valid as those that use the formal properties of δ. The following preliminary result gives us this assurance.

These examples show how convenient transition graphs are for working with finite automata. While it is possible to base all arguments strictly on the properties of the transition function and its extension through (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$
$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

the results are hard to follow. In our discussion, we use graphs, which are more intuitive, as far as possible. To do so, we must, of course, have some assurance that we are not misled by the representation and that arguments based on graphs are as valid as those that use the formal properties of δ. The following preliminary result gives us this assurance.

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n + 1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n + 1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ■

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n + 1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ■

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ■

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n + 1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \qquad (4)$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ■

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ■

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n + 1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ■

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \qquad (4)$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ■

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n + 1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n + 1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ■

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n + 1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \qquad (4)$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ■

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Theorem 2.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof. This claim is fairly obvious from an examination of such simple cases as **Example 2.1**. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leqslant n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus, there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (4), thus completing the proof. ∎

Again, the result of the theorem is so intuitively obvious that a formal proof seems unnecessary. We went through the details for two reasons. The first is that it is a simple, yet typical example of an inductive proof in connection with automata. The second is that the result will be used over and over, so stating and proving it as a theorem lets us argue quite confidently using graphs. This makes our examples and proofs more transparent than they would be if we used the properties of δ^*.

Again, the result of the theorem is so intuitively obvious that a formal proof seems unnecessary. We went through the details for two reasons. The first is that it is a simple, yet typical example of an inductive proof in connection with automata. The second is that the result will be used over and over, so stating and proving it as a theorem lets us argue quite confidently using graphs. This makes our examples and proofs more transparent than they would be if we used the properties of δ^*.

Again, the result of the theorem is so intuitively obvious that a formal proof seems unnecessary. We went through the details for two reasons. The first is that it is a simple, yet typical example of an inductive proof in connection with automata. The second is that the result will be used over and over, so stating and proving it as a theorem lets us argue quite confidently using graphs. This makes our examples and proofs more transparent than they would be if we used the properties of δ^*.

Again, the result of the theorem is so intuitively obvious that a formal proof seems unnecessary. We went through the details for two reasons. The first is that it is a simple, yet typical example of an inductive proof in connection with automata. The second is that the result will be used over and over, so stating and proving it as a theorem lets us argue quite confidently using graphs. This makes our examples and proofs more transparent than they would be if we used the properties of δ^*.

Again, the result of the theorem is so intuitively obvious that a formal proof seems unnecessary. We went through the details for two reasons. The first is that it is a simple, yet typical example of an inductive proof in connection with automata. The second is that the result will be used over and over, so stating and proving it as a theorem lets us argue quite confidently using graphs. This makes our examples and proofs more transparent than they would be if we used the properties of δ^*.

Again, the result of the theorem is so intuitively obvious that a formal proof seems unnecessary. We went through the details for two reasons. The first is that it is a simple, yet typical example of an inductive proof in connection with automata. The second is that the result will be used over and over, so stating and proving it as a theorem lets us argue quite confidently using graphs. This makes our examples and proofs more transparent than they would be if we used the properties of δ^*.

Again, the result of the theorem is so intuitively obvious that a formal proof seems unnecessary. We went through the details for two reasons. The first is that it is a simple, yet typical example of an inductive proof in connection with automata. The second is that the result will be used over and over, so stating and proving it as a theorem lets us argue quite confidently using graphs. This makes our examples and proofs more transparent than they would be if we used the properties of δ^*.

Again, the result of the theorem is so intuitively obvious that a formal proof seems unnecessary. We went through the details for two reasons. The first is that it is a simple, yet typical example of an inductive proof in connection with automata. The second is that the result will be used over and over, so stating and proving it as a theorem lets us argue quite confidently using graphs. This makes our examples and proofs more transparent than they would be if we used the properties of δ^*.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in the Figure

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

is equivalent to the following Figure

Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in the Figure

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

is equivalent to the following Figure

Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in the Figure

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

is equivalent to the following Figure

Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in the Figure

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

is equivalent to the following Figure

Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in the Figure

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

is equivalent to the following Figure

Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in the Figure

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

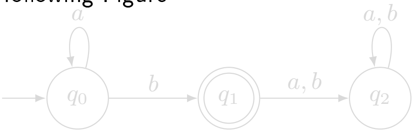
is equivalent to the following Figure

Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in the Figure

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

is equivalent to the following Figure

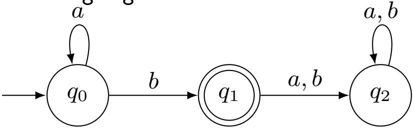


Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in the Figure

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

is equivalent to the following Figure

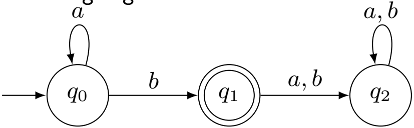


Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in the Figure

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

is equivalent to the following Figure

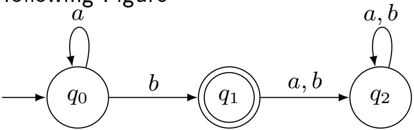


Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in the Figure

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

is equivalent to the following Figure

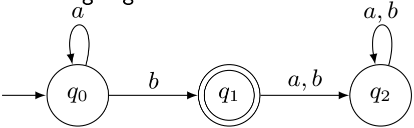


Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in the Figure

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

is equivalent to the following Figure



Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

It is apparent from this example that a DFA can easily be implemented as a computer program; for example, as a simple table-lookup or as a sequence of if statements. The best implementation or representation depends on the specific application. Transition graphs are very convenient for the kinds of arguments we want to make here, so we use them in most of our discussions.

In constructing automata for languages defined informally, we employ reasoning similar to that for programming in higher-level languages. But the programming of a DFA is tedious and sometimes conceptually complicated by the fact that such an automaton has few powerful features.

It is apparent from this example that a DFA can easily be implemented as a computer program; for example, as a simple table-lookup or as a sequence of if statements. The best implementation or representation depends on the specific application. Transition graphs are very convenient for the kinds of arguments we want to make here, so we use them in most of our discussions.

In constructing automata for languages defined informally, we employ reasoning similar to that for programming in higher-level languages. But the programming of a DFA is tedious and sometimes conceptually complicated by the fact that such an automaton has few powerful features.

It is apparent from this example that a DFA can easily be implemented as a computer program; for example, as a simple table-lookup or as a sequence of if statements. The best implementation or representation depends on the specific application. Transition graphs are very convenient for the kinds of arguments we want to make here, so we use them in most of our discussions.

In constructing automata for languages defined informally, we employ reasoning similar to that for programming in higher-level languages. But the programming of a DFA is tedious and sometimes conceptually complicated by the fact that such an automaton has few powerful features.

It is apparent from this example that a DFA can easily be implemented as a computer program; for example, as a simple table-lookup or as a sequence of if statements. The best implementation or representation depends on the specific application. Transition graphs are very convenient for the kinds of arguments we want to make here, so we use them in most of our discussions.

In constructing automata for languages defined informally, we employ reasoning similar to that for programming in higher-level languages. But the programming of a DFA is tedious and sometimes conceptually complicated by the fact that such an automaton has few powerful features.

It is apparent from this example that a DFA can easily be implemented as a computer program; for example, as a simple table-lookup or as a sequence of if statements. The best implementation or representation depends on the specific application. Transition graphs are very convenient for the kinds of arguments we want to make here, so we use them in most of our discussions.

In constructing automata for languages defined informally, we employ reasoning similar to that for programming in higher-level languages. But the programming of a DFA is tedious and sometimes conceptually complicated by the fact that such an automaton has few powerful features.

It is apparent from this example that a DFA can easily be implemented as a computer program; for example, as a simple table-lookup or as a sequence of if statements. The best implementation or representation depends on the specific application. Transition graphs are very convenient for the kinds of arguments we want to make here, so we use them in most of our discussions.

In constructing automata for languages defined informally, we employ reasoning similar to that for programming in higher-level languages. But the programming of a DFA is tedious and sometimes conceptually complicated by the fact that such an automaton has few powerful features.

It is apparent from this example that a DFA can easily be implemented as a computer program; for example, as a simple table-lookup or as a sequence of if statements. The best implementation or representation depends on the specific application. Transition graphs are very convenient for the kinds of arguments we want to make here, so we use them in most of our discussions.

In constructing automata for languages defined informally, we employ reasoning similar to that for programming in higher-level languages. But the programming of a DFA is tedious and sometimes conceptually complicated by the fact that such an automaton has few powerful features.

It is apparent from this example that a DFA can easily be implemented as a computer program; for example, as a simple table-lookup or as a sequence of if statements. The best implementation or representation depends on the specific application. Transition graphs are very convenient for the kinds of arguments we want to make here, so we use them in most of our discussions.

In constructing automata for languages defined informally, we employ reasoning similar to that for programming in higher-level languages. But the programming of a DFA is tedious and sometimes conceptually complicated by the fact that such an automaton has few powerful features.

It is apparent from this example that a DFA can easily be implemented as a computer program; for example, as a simple table-lookup or as a sequence of if statements. The best implementation or representation depends on the specific application. Transition graphs are very convenient for the kinds of arguments we want to make here, so we use them in most of our discussions.

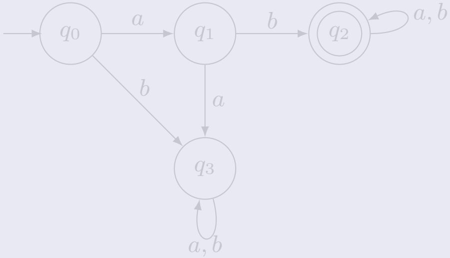
In constructing automata for languages defined informally, we employ reasoning similar to that for programming in higher-level languages. But the programming of a DFA is tedious and sometimes conceptually complicated by the fact that such an automaton has few powerful features.

Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states; an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.

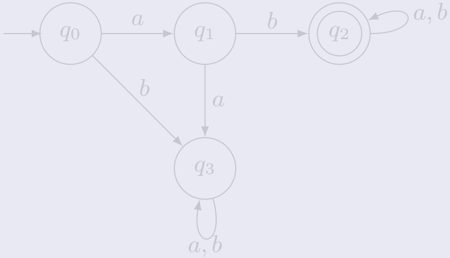
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



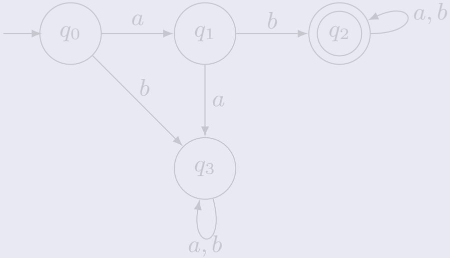
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



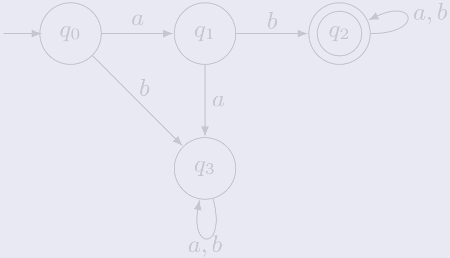
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



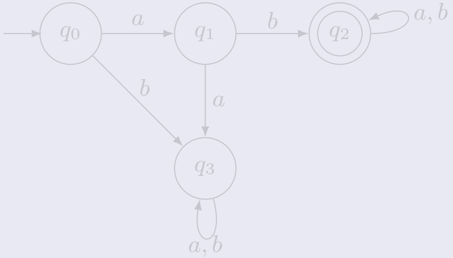
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



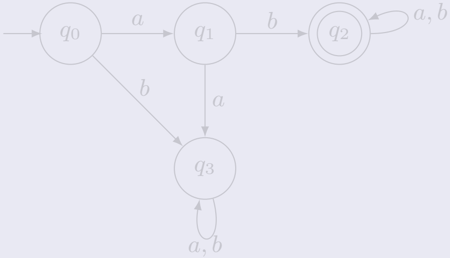
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



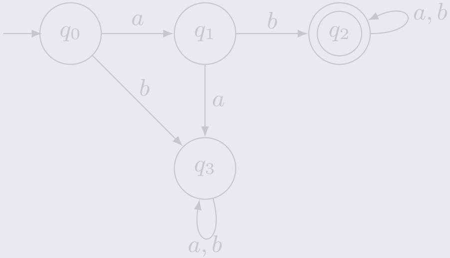
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



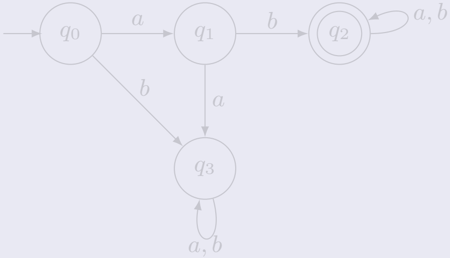
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



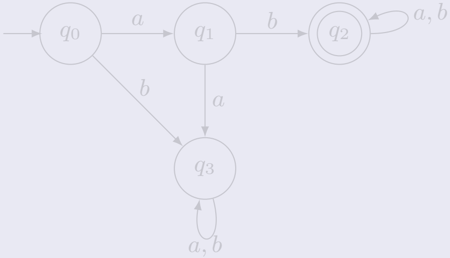
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



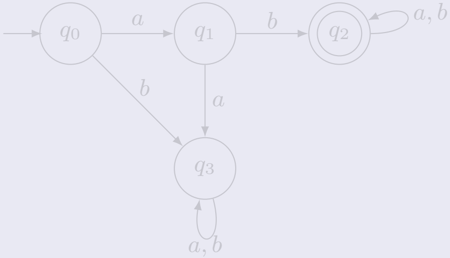
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



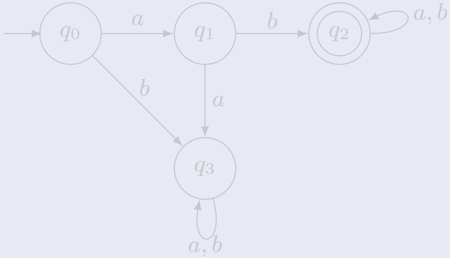
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



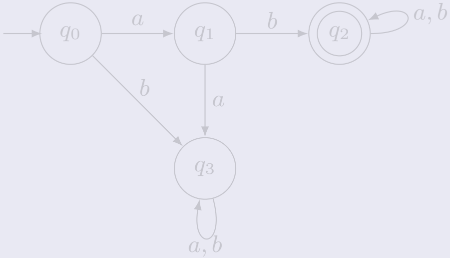
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



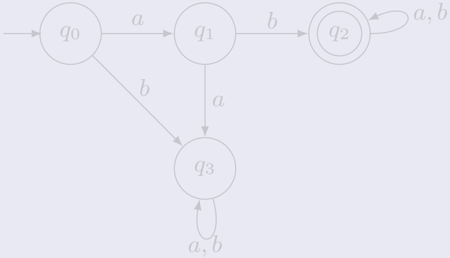
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



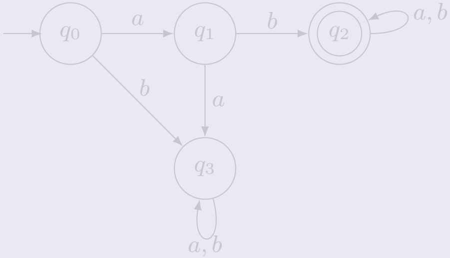
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



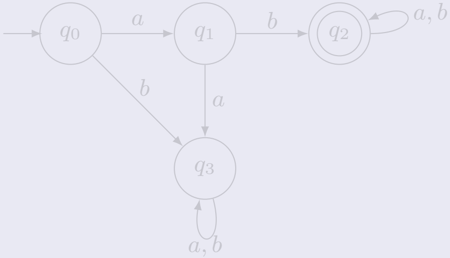
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



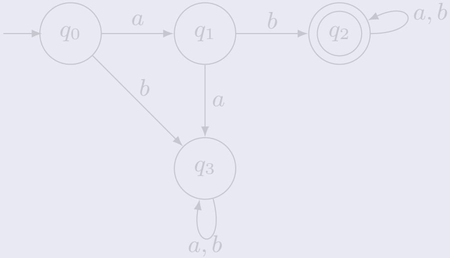
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



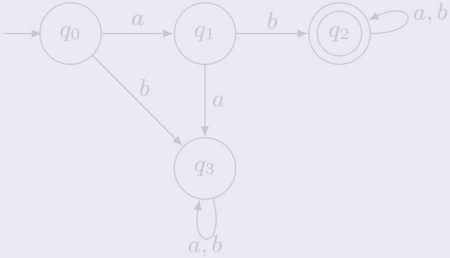
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



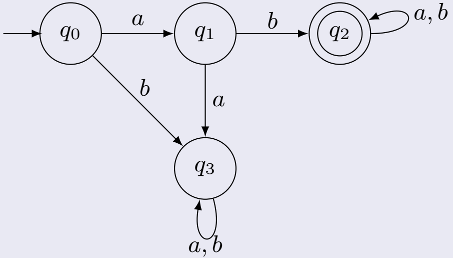
Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



Example 2.3

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab. The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is the letter a and the second is the letter b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not the letter a or the second one is not the letter b, the automaton enters the nonfinal trap state. The simple solution is shown in the following Figure.



Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4

Find a DFA that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.

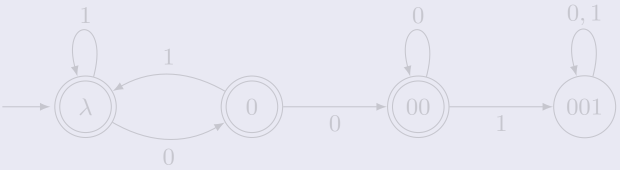
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



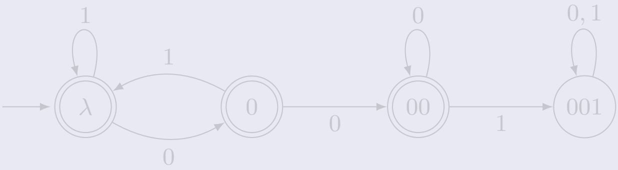
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



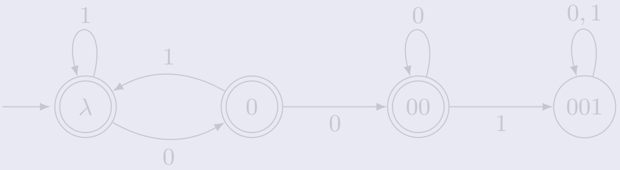
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



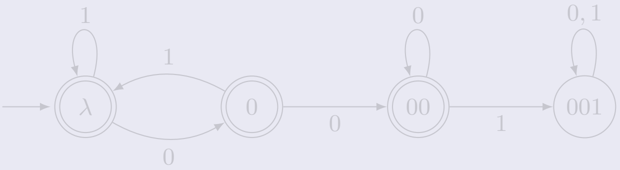
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



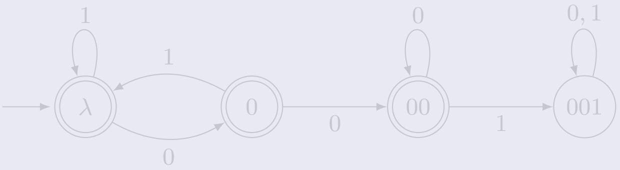
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



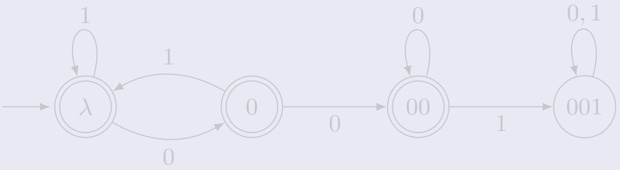
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



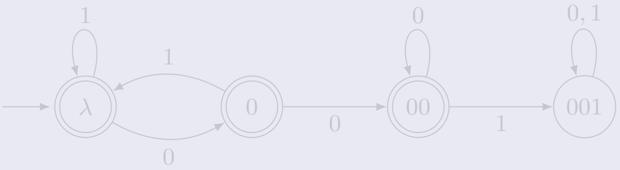
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



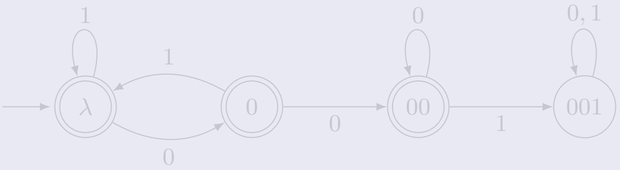
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



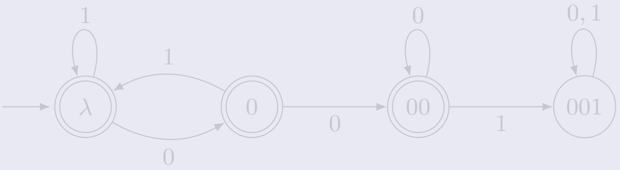
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



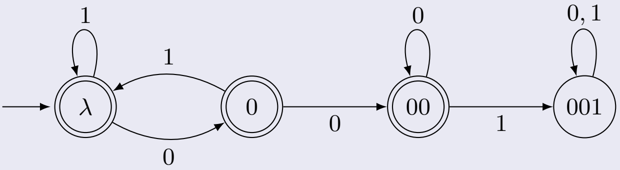
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



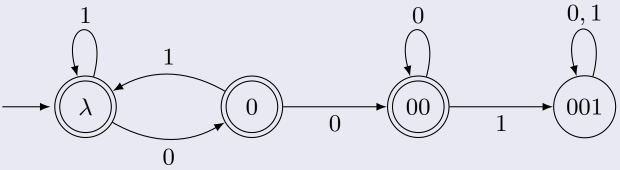
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



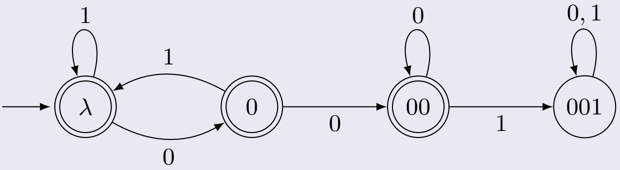
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



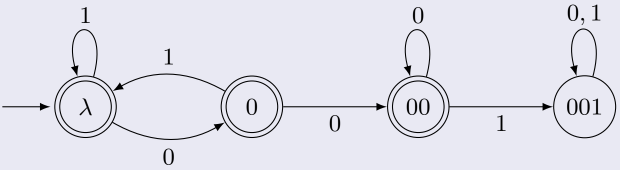
We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Example 2.4 (continuation)

If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the DFA in the state 00. A complete solution is shown in the following Figure.



We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

2.1 Deterministic Finite Accepters: Regular Languages

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa \colon w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

2.1 Deterministic Finite Accepters: Regular Languages

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa \colon w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa \colon w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa \colon w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa \colon w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa \colon w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa \colon w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa \colon w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa \colon w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We shall call such a set of languages a *family*. The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.2

A language L is called *regular* if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5

Show that the language

$$L = \{awa \colon w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a DFA for it. The construction of a DFA for this language is similar to Example 2.3, but a little more complicated. What this DFA must do is check whether a string begins and ends with the letter a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string.

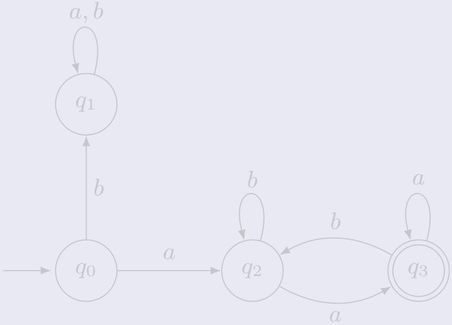
Example 2.5 (continuation)

This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.

Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.5 (continuation)

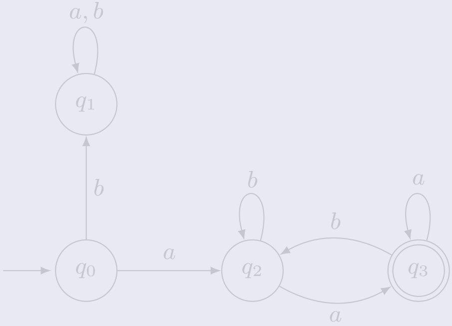
This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.



Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.5 (continuation)

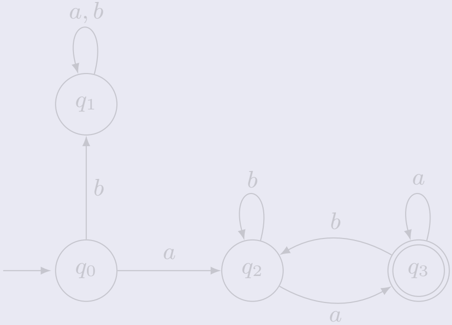
This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.



Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.5 (continuation)

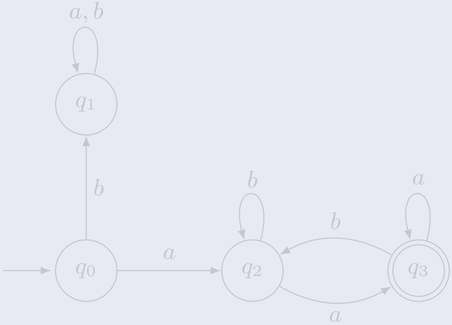
This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.



Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.5 (continuation)

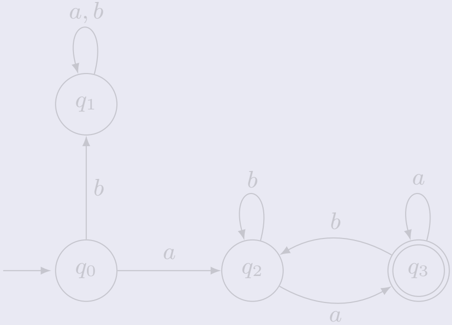
This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.



Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.5 (continuation)

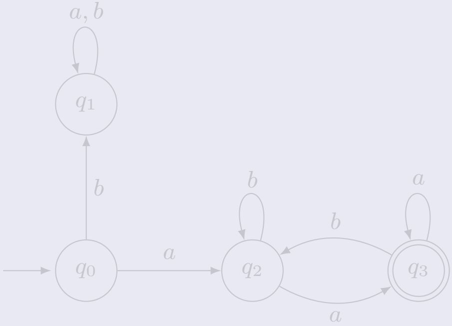
This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.



Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.5 (continuation)

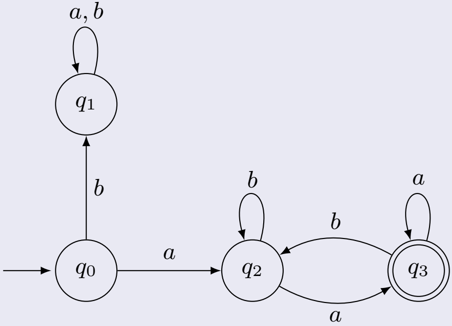
This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.



Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.5 (continuation)

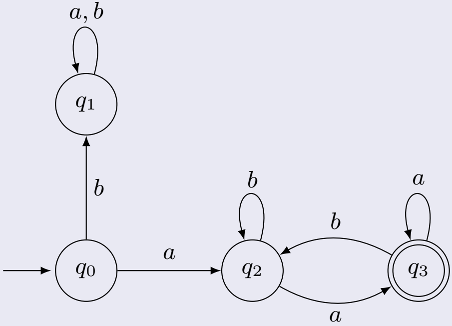
This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.



Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.5 (continuation)

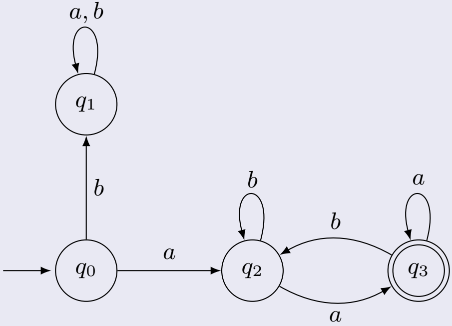
This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.



Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.5 (continuation)

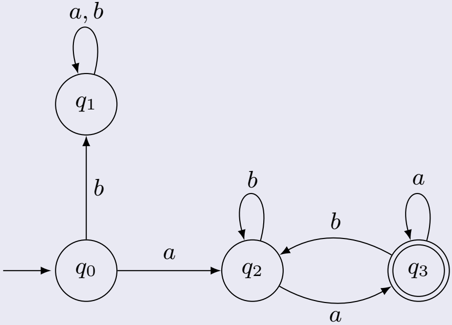
This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.



Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.5 (continuation)

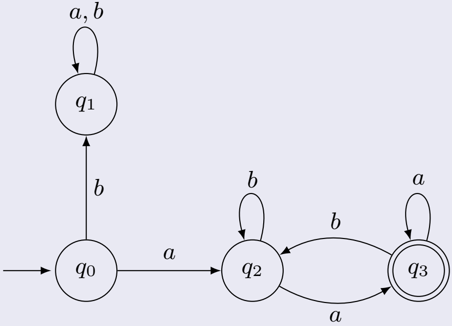
This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.



Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.5 (continuation)

This difficulty is overcome by simply putting the DFA into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the DFA out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in the Figure below.



Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the DFA accepts a string if and only if it begins and ends with the letter a. Since we have constructed a DFA for the language, we can claim that, by definition, the language is regular.

Example 2.6

Let L be the language in Example 2.5:

$$L = \{awa : w \in \{a, b\}^*\}.$$

Show that L^2 is regular. Again we show that the language is regular by constructing a DFA for it. We can write an explicit expression for L^2, namely,

$$L^2 = \{aw_1aaw_2a : w_1, w_2 \in \{a, b\}^*\}.$$

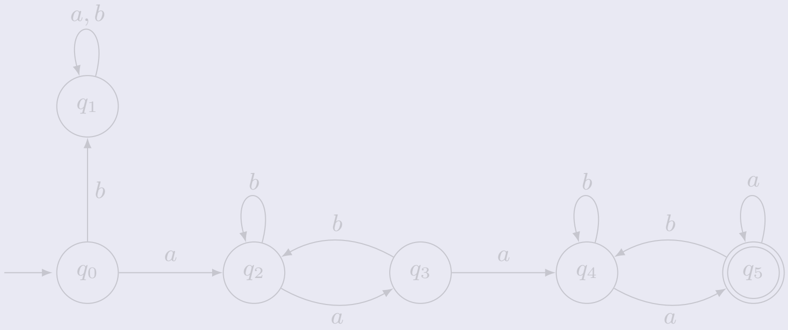
Example 2.6

Let L be the language in Example 2.5:

$$L = \{awa\colon w \in \{a,b\}^*\}\,.$$

Show that L^2 is regular. Again we show that the language is regular by constructing a DFA for it. We can write an explicit expression for L^2, namely,

$$L^2 = \{aw_1aaw_2a\colon w_1, w_2 \in \{a,b\}^*\}\,.$$



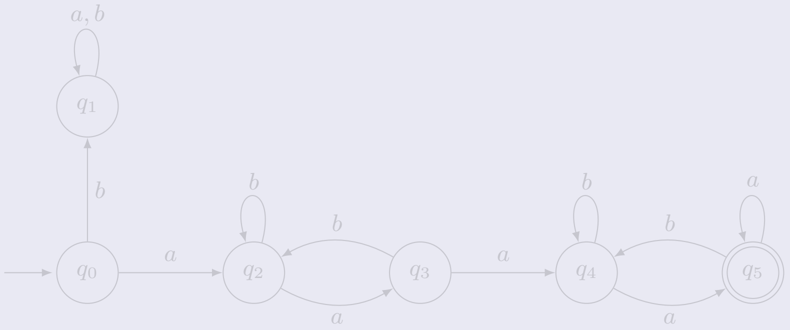
Example 2.6

Let L be the language in Example 2.5:

$$L = \{awa\colon w \in \{a,b\}^*\}.$$

Show that L^2 is regular. Again we show that the language is regular by constructing a DFA for it. We can write an explicit expression for L^2, namely,

$$L^2 = \{aw_1aaw_2a\colon w_1, w_2 \in \{a,b\}^*\}.$$



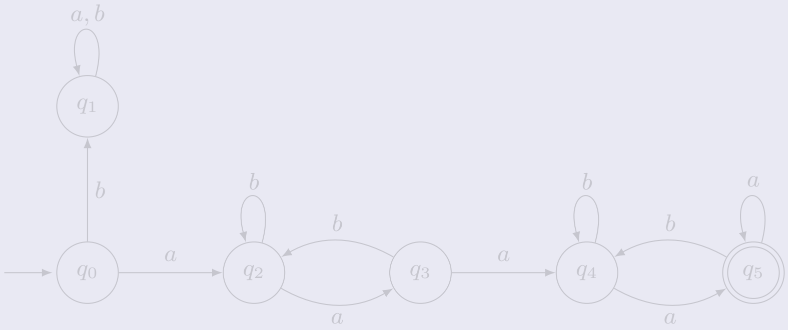
Example 2.6

Let L be the language in Example 2.5:

$$L = \{awa\colon w \in \{a,b\}^*\}.$$

Show that L^2 is regular. Again we show that the language is regular by constructing a DFA for it. We can write an explicit expression for L^2, namely,

$$L^2 = \{aw_1aaw_2a\colon w_1, w_2 \in \{a,b\}^*\}.$$



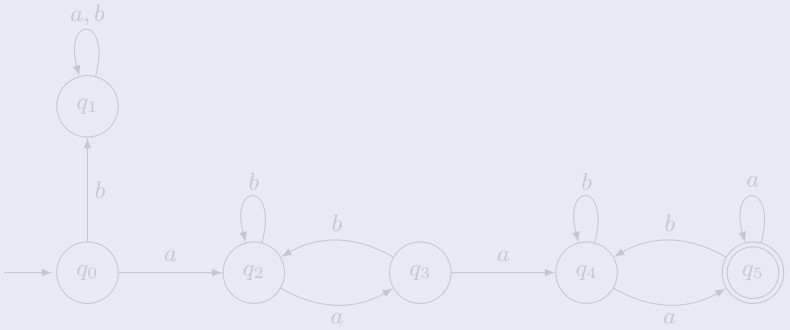
Example 2.6

Let L be the language in Example 2.5:

$$L = \{awa\colon w \in \{a,b\}^*\}.$$

Show that L^2 is regular. Again we show that the language is regular by constructing a DFA for it. We can write an explicit expression for L^2, namely,

$$L^2 = \{aw_1aaw_2a\colon w_1, w_2 \in \{a,b\}^*\}.$$



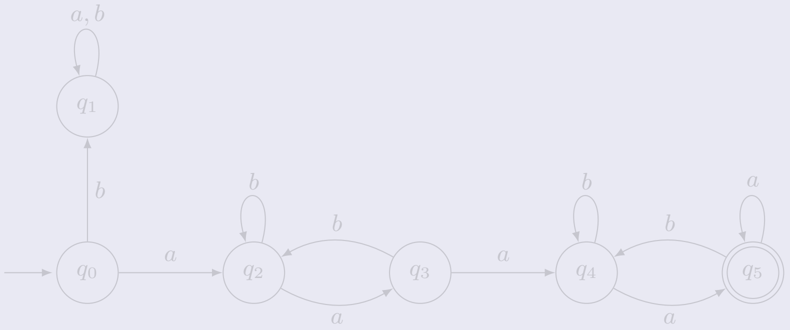
Example 2.6

Let L be the language in Example 2.5:

$$L = \{awa \colon w \in \{a, b\}^*\}.$$

Show that L^2 is regular. Again we show that the language is regular by constructing a DFA for it. We can write an explicit expression for L^2, namely,

$$L^2 = \{aw_1aaw_2a \colon w_1, w_2 \in \{a, b\}^*\}.$$



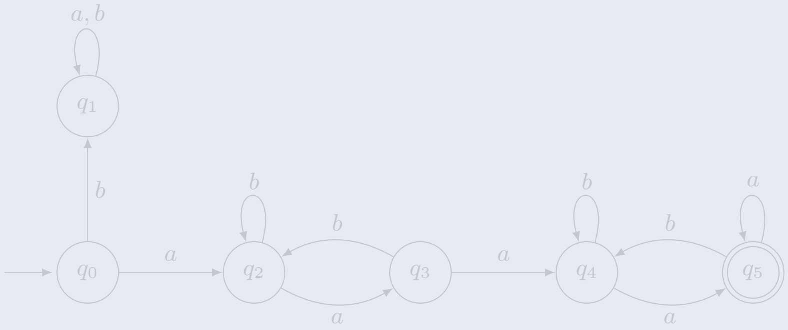
Example 2.6

Let L be the language in Example 2.5:

$$L = \{awa\colon w \in \{a,b\}^*\}.$$

Show that L^2 is regular. Again we show that the language is regular by constructing a DFA for it. We can write an explicit expression for L^2, namely,

$$L^2 = \{aw_1aaw_2a\colon w_1, w_2 \in \{a,b\}^*\}.$$



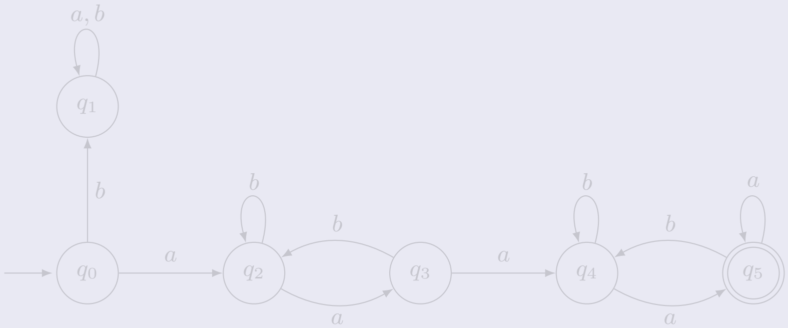
Example 2.6

Let L be the language in Example 2.5:

$$L = \{awa\colon w \in \{a,b\}^*\}.$$

Show that L^2 is regular. Again we show that the language is regular by constructing a DFA for it. We can write an explicit expression for L^2, namely,

$$L^2 = \{aw_1aaw_2a\colon w_1, w_2 \in \{a,b\}^*\}.$$



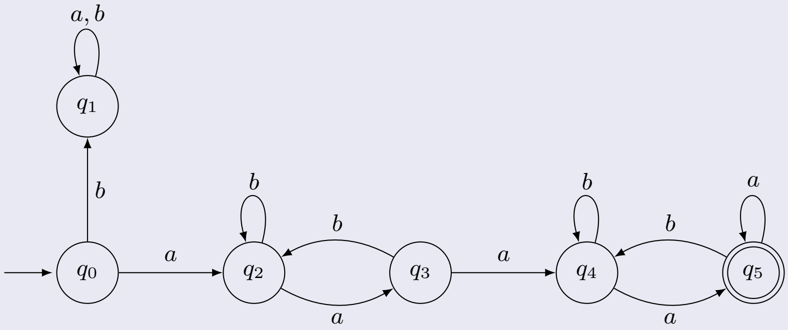
Example 2.6

Let L be the language in Example 2.5:

$$L = \{awa\colon w \in \{a,b\}^*\}.$$

Show that L^2 is regular. Again we show that the language is regular by constructing a DFA for it. We can write an explicit expression for L^2, namely,

$$L^2 = \{aw_1aaw_2a\colon w_1, w_2 \in \{a,b\}^*\}.$$



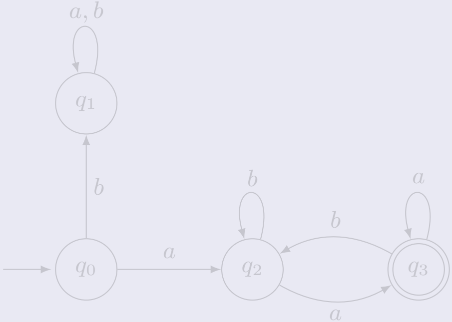
Example 2.6 (continuation)

Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.

This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

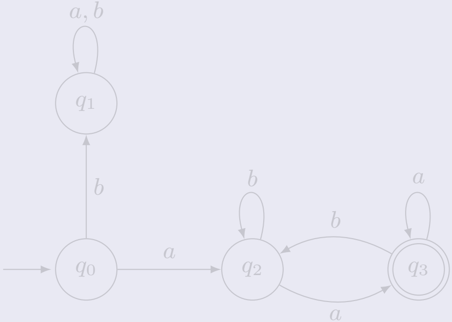
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

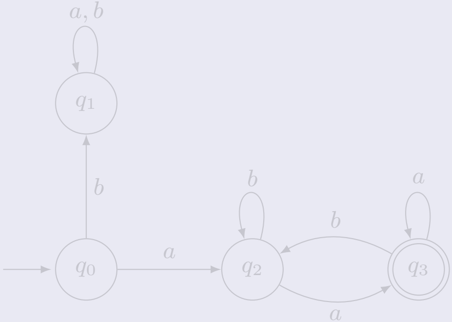
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

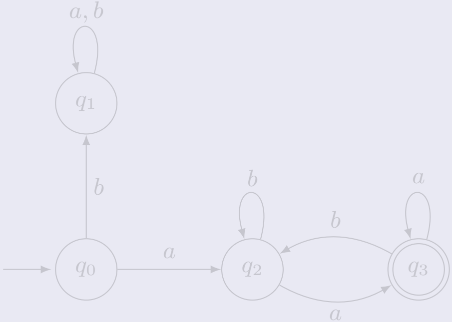
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

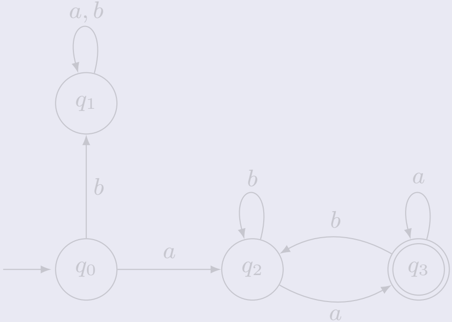
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

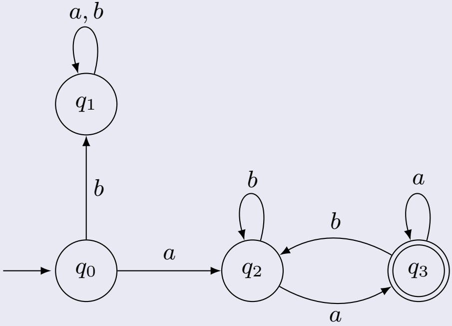
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

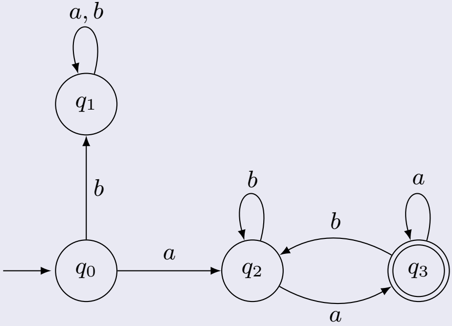
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

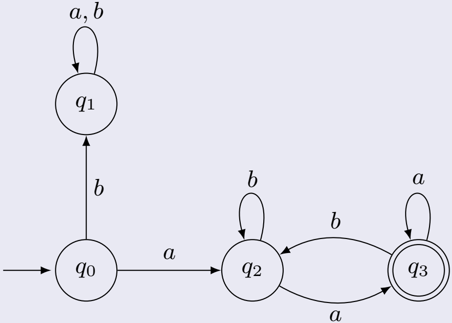
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

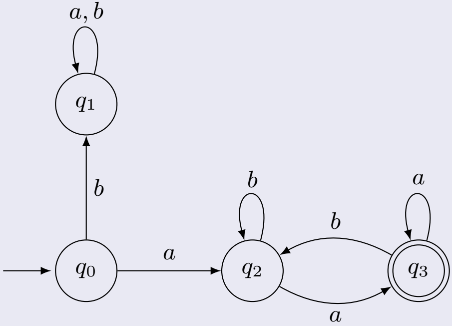
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

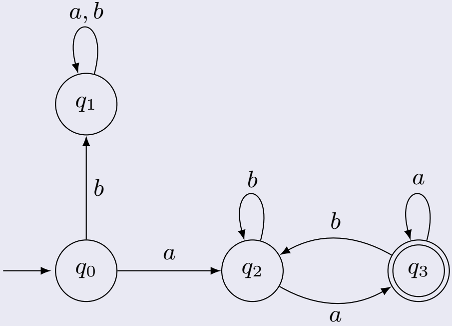
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

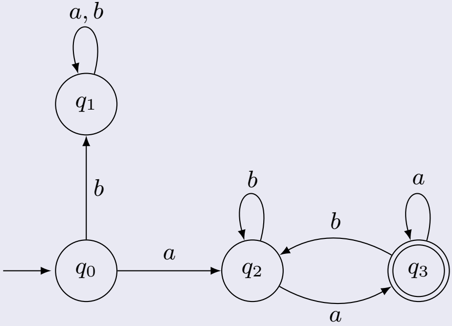
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

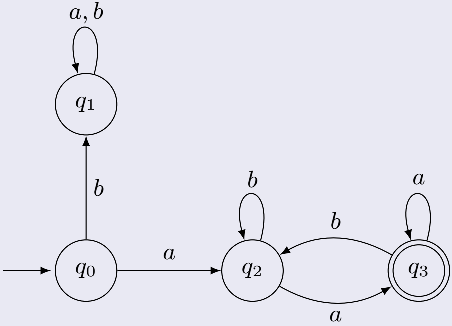
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

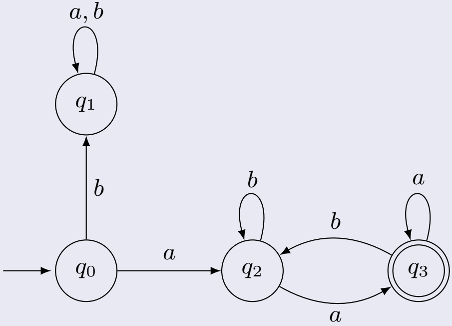
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

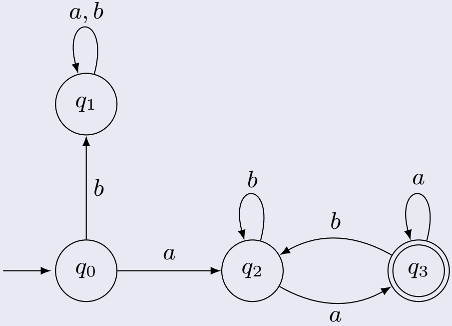
Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

Therefore, we need a DFA that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in the Figure of Example 2.5 can be used as a starting point, but the vertex q_3 has to be modified.



This state can no longer be final because, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part.

Example 2.6 (continuation)

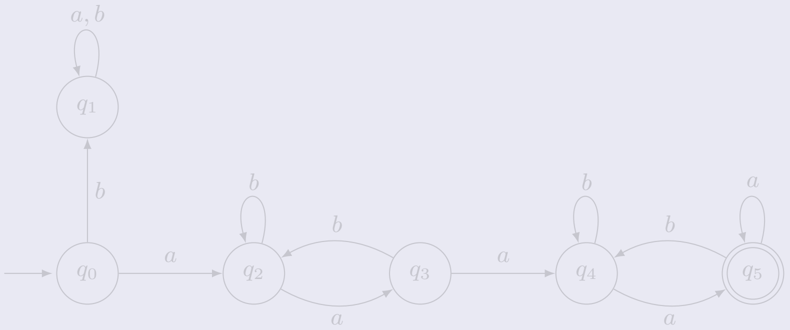
We can do this by making $\delta(q_3, a) = q_4$. The complete solution is in the Figure below.

This DFA accepts L^2, which is therefore regular.

The last example suggests the conjecture that if a language L is regular, so are $L^2, L^3, \ldots$. We shall see later that this is indeed correct.

Example 2.6 (continuation)

We can do this by making $\delta(q_3, a) = q_4$. The complete solution is in the Figure below.

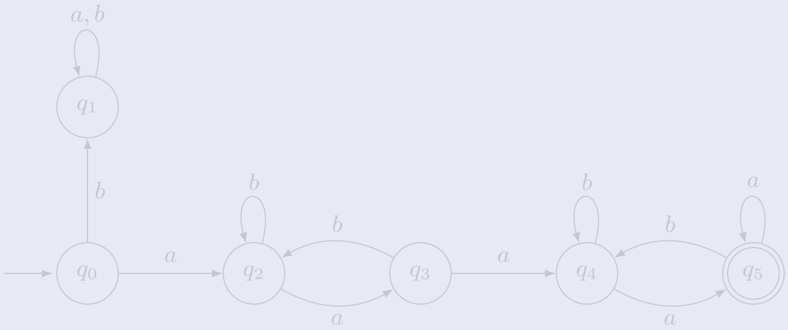


This DFA accepts L^2, which is therefore regular.

The last example suggests the conjecture that if a language L is regular, so are $L^2, L^3, \ldots$. We shall see later that this is indeed correct.

Example 2.6 (continuation)

We can do this by making $\delta(q_3, a) = q_4$. The complete solution is in the Figure below.

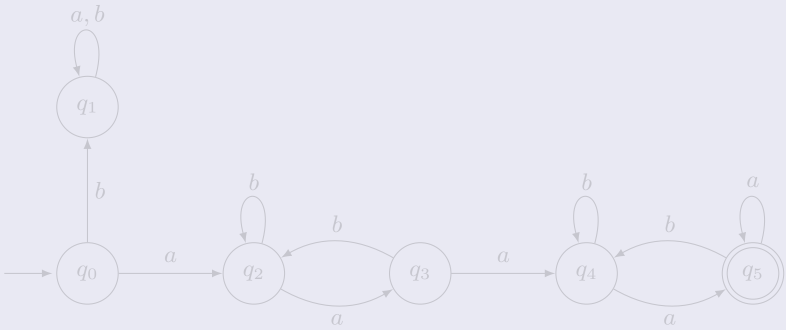


This DFA accepts L^2, which is therefore regular.

The last example suggests the conjecture that if a language L is regular, so are $L^2, L^3, \ldots$. We shall see later that this is indeed correct.

Example 2.6 (continuation)

We can do this by making $\delta(q_3, a) = q_4$. The complete solution is in the Figure below.

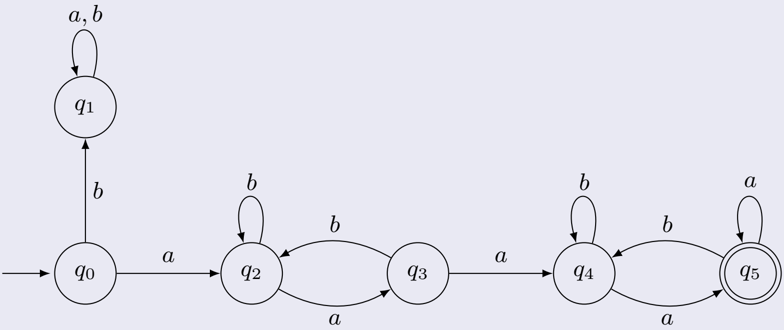


This DFA accepts L^2, which is therefore regular.

The last example suggests the conjecture that if a language L is regular, so are $L^2, L^3, \ldots$. We shall see later that this is indeed correct.

Example 2.6 (continuation)

We can do this by making $\delta(q_3, a) = q_4$. The complete solution is in the Figure below.

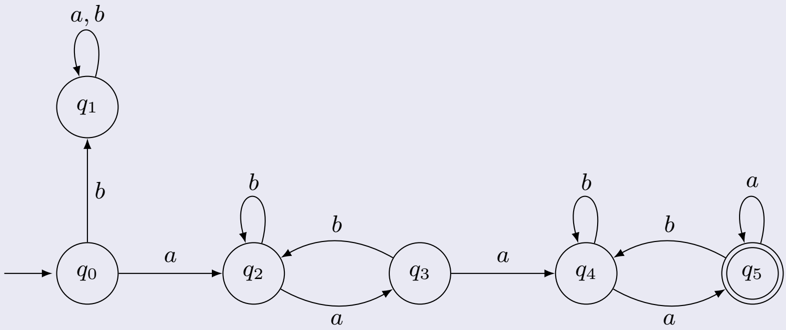


This DFA accepts L^2, which is therefore regular.

The last example suggests the conjecture that if a language L is regular, so are $L^2, L^3, \ldots$. We shall see later that this is indeed correct.

Example 2.6 (continuation)

We can do this by making $\delta(q_3, a) = q_4$. The complete solution is in the Figure below.

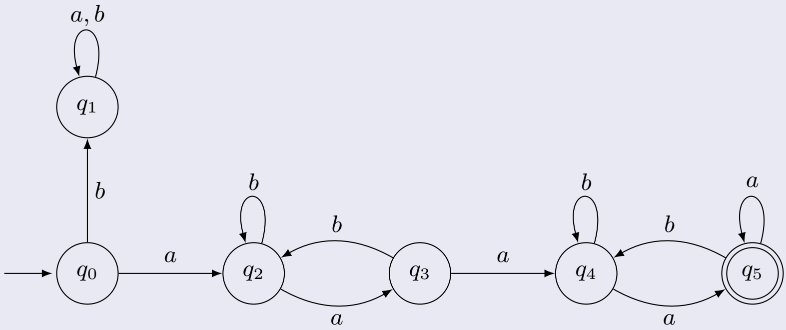


This DFA accepts L^2, which is therefore regular.

The last example suggests the conjecture that if a language L is regular, so are $L^2, L^3, \ldots$. We shall see later that this is indeed correct.

Example 2.6 (continuation)

We can do this by making $\delta(q_3, a) = q_4$. The complete solution is in the Figure below.

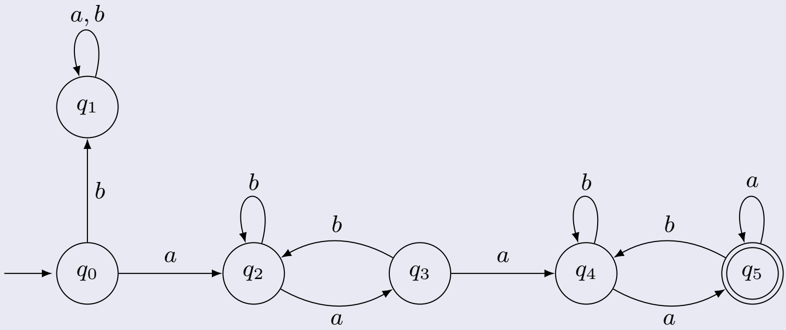


This DFA accepts L^2, which is therefore regular.

The last example suggests the conjecture that if a language L is regular, so are $L^2, L^3, \ldots$. We shall see later that this is indeed correct.

Example 2.6 (continuation)

We can do this by making $\delta(q_3, a) = q_4$. The complete solution is in the Figure below.



This DFA accepts L^2, which is therefore regular.

The last example suggests the conjecture that if a language L is regular, so are $L^2, L^3, \ldots$. We shall see later that this is indeed correct.

Thank You for attention!