Formal Languages, Automata and Codes

Oleg Gutik

Lecture 9

3.2 Connections between Regular Expressions and Regular Languages

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.
   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of Definition 3.2, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of Definition 3.2, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.
   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of Definition 3.2, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of Definition 3.2, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.
   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

3.2 Connections between Regular Expressions and Regular Languages

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.
   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.
   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.
   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.
   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.
   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

As the terminology suggests, the connection between regular languages and regular expressions is a close one. The two concepts are essentially the same; for every regular language there is a regular expression, and for every regular expression there is a regular language. We will show this in two parts.

Regular Expressions Denote Regular Languages

We first show that if r is a regular expression, then $L(r)$ is a regular language. Our definition says that a language is regular if it is accepted by some DFA. Because of the equivalence of NFA's and DFA's, a language is also regular if it is accepted by some NFA. We now show that if we have any regular expression r, we can construct an NFA that accepts $L(r)$. The construction for this relies on the recursive definition for $L(r)$. We first construct simple automata for parts (1), (2), and (3) of **Definition 3.2**, then show how they can be combined to implement the more complicated parts (4), (5), and (7).

Definition 3.2

The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set,
2. λ is a regular expression denoting $\{\lambda\}$,
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.
   If r_1 and r_2 are regular expressions, then
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.

(a) (b) (c)

Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.

(a) (b) (c)

Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.

(a) (b) (c)

Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.

(a) (b) (c)

Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.

(a) (b) (c)

Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.

(a) (b) (c)

Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.

(a) (b) (c)

Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.

(a) (b) (c)

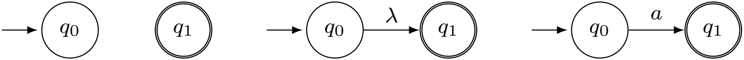
Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.



(a) (b) (c)

Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

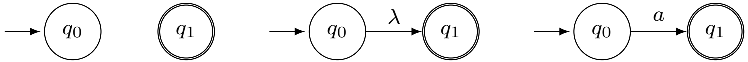
In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

3.2 Regular Expressions and Regular Languages

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.



(a) (b) (c)

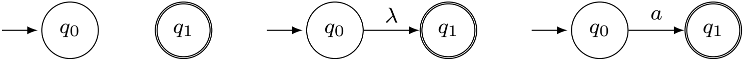
Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.



(a) (b) (c)

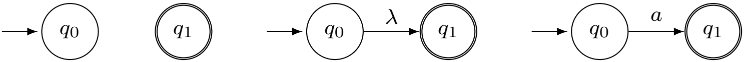
Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.



(a) (b) (c)

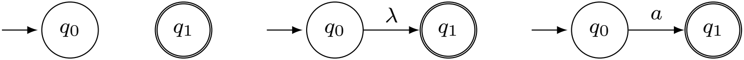
Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

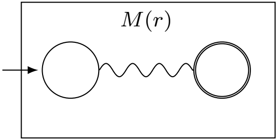
Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.



(a) (b) (c)

Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.

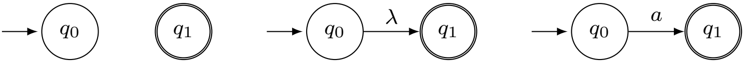


In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

Theorem 3.1

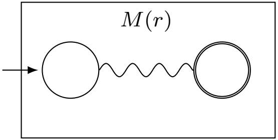
Let r be a regular expression. Then there exists some nondeterministic finite accepter that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof. We begin with automata that accept the languages for the simple regular expressions $\varnothing$, λ, and $a \in \Sigma$. These are shown in Figure (a), (b), and (c), respectively.



(a) (b) (c)

Assume now that we have automata $M(r_1)$ and $M(r_2)$ that accept languages denoted by regular expressions r_1 and r_2, respectively. We need not explicitly construct these automata, but may represent them schematically, as in the Figure.



In this scheme, the graph vertex at the left represents the initial state, the one on the right the final state.

In previous lectutes we claim that for every NFA there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, $r_1 r_2$, and r_1^*. The constructions are shown in the following three Figures. As indicated in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.

Automaton for $L(r_1 + r_2)$.

In previous lectutes we claim that for every NFA there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, $r_1 r_2$, and r_1^*. The constructions are shown in the following three Figures. As indicated in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.

Automaton for $L(r_1 + r_2)$.

In previous lectutes we claim that for every NFA there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, $r_1 r_2$, and r_1^*. The constructions are shown in the following three Figures. As indicated in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.

Automaton for $L(r_1 + r_2)$.

In previous lectutes we claim that for every NFA there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, $r_1 r_2$, and r_1^*. The constructions are shown in the following three Figures. As indicated in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.

Automaton for $L(r_1 + r_2)$.

In previous lectutes we claim that for every NFA there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, $r_1 r_2$, and r_1^*. The constructions are shown in the following three Figures. As indicated in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.

Automaton for $L(r_1 + r_2)$.

In previous lectutes we claim that for every NFA there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, $r_1 r_2$, and r_1^*. The constructions are shown in the following three Figures. As indicated in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.

Automaton for $L(r_1 + r_2)$.

In previous lectutes we claim that for every NFA there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, $r_1 r_2$, and r_1^*. The constructions are shown in the following three Figures. As indicated in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.

Automaton for $L(r_1 + r_2)$.

In previous lectutes we claim that for every NFA there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, $r_1 r_2$, and r_1^*. The constructions are shown in the following three Figures. As indicated in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.

Automaton for $L(r_1 + r_2)$.

In previous lectutes we claim that for every NFA there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, $r_1 r_2$, and r_1^*. The constructions are shown in the following three Figures. As indicated in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.

Automaton for $L(r_1 + r_2)$.

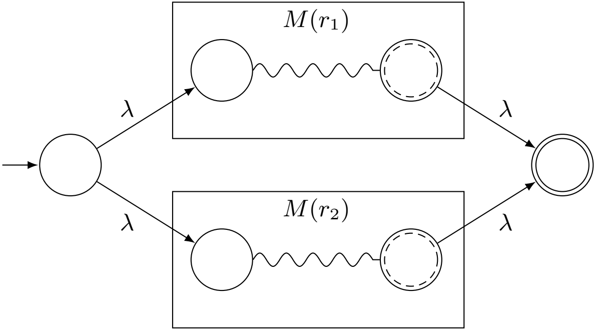
3.2 Regular Expressions and Regular Languages

In previous lectutes we claim that for every NFA there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, $r_1 r_2$, and r_1^*. The constructions are shown in the following three Figures. As indicated in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.

Automaton for $L(r_1 + r_2)$.

3.2 Regular Expressions and Regular Languages

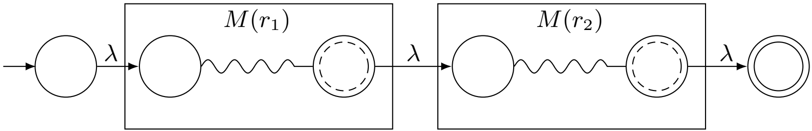
In previous lectutes we claim that for every NFA there is an equivalent one with a single final state, so we lose nothing in assuming that there is only one final state. With $M(r_1)$ and $M(r_2)$ represented in this way, we then construct automata for the regular expressions $r_1 + r_2$, $r_1 r_2$, and r_1^*. The constructions are shown in the following three Figures. As indicated in the drawings, the initial and final states of the constituent machines lose their status and are replaced by new initial and final states. By stringing together several such steps, we can build automata for arbitrary complex regular expressions.



Automaton for $L(r_1 + r_2)$.

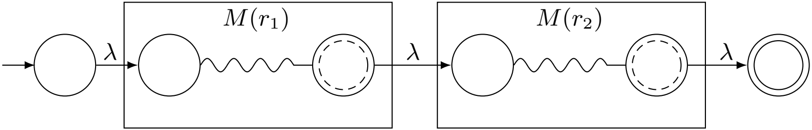
Automaton for $L(r_1 r_2)$.

Automaton for $L(r_1^*)$.

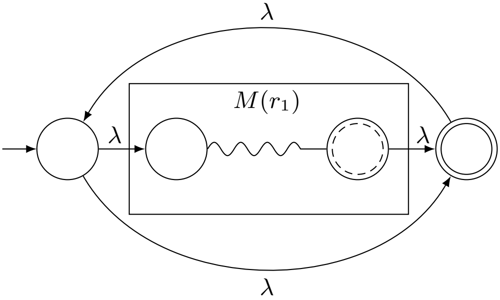


Automaton for $L(r_1 r_2)$.

Automaton for $L(r_1^*)$.



Automaton for $L(r_1 r_2)$.



Automaton for $L(r_1^*)$.

It should be clear from the interpretation of the graphs in above three that this construction works. To argue more rigorously, we can give a formal method for constructing the states and transitions of the combined machine from the states and transitions of the parts, then prove by induction on the number of operators that the construction yields an automaton that accepts the language denoted by any particular regular expression. We will not belabor this point, as it is reasonably obvious that the results are always correct. ■

It should be clear from the interpretation of the graphs in above three that this construction works. To argue more rigorously, we can give a formal method for constructing the states and transitions of the combined machine from the states and transitions of the parts, then prove by induction on the number of operators that the construction yields an automaton that accepts the language denoted by any particular regular expression. We will not belabor this point, as it is reasonably obvious that the results are always correct. ■

It should be clear from the interpretation of the graphs in above three that this construction works. To argue more rigorously, we can give a formal method for constructing the states and transitions of the combined machine from the states and transitions of the parts, then prove by induction on the number of operators that the construction yields an automaton that accepts the language denoted by any particular regular expression. We will not belabor this point, as it is reasonably obvious that the results are always correct. ■

It should be clear from the interpretation of the graphs in above three that this construction works. To argue more rigorously, we can give a formal method for constructing the states and transitions of the combined machine from the states and transitions of the parts, then prove by induction on the number of operators that the construction yields an automaton that accepts the language denoted by any particular regular expression. We will not belabor this point, as it is reasonably obvious that the results are always correct. ■

It should be clear from the interpretation of the graphs in above three that this construction works. To argue more rigorously, we can give a formal method for constructing the states and transitions of the combined machine from the states and transitions of the parts, then prove by induction on the number of operators that the construction yields an automaton that accepts the language denoted by any particular regular expression. We will not belabor this point, as it is reasonably obvious that the results are always correct. ■

It should be clear from the interpretation of the graphs in above three that this construction works. To argue more rigorously, we can give a formal method for constructing the states and transitions of the combined machine from the states and transitions of the parts, then prove by induction on the number of operators that the construction yields an automaton that accepts the language denoted by any particular regular expression. We will not belabor this point, as it is reasonably obvious that the results are always correct. ■

Example 3.7

Find an NFA that accepts $L(r)$, where

$$r = (a + bb)^*(ba^* + \lambda).$$

Automata for $(a + bb)$ and $(ba^* + \lambda)$, constructed directly from first principles, are given in the following Figure.

(a) M_1 accepts $L(a + bb)$.

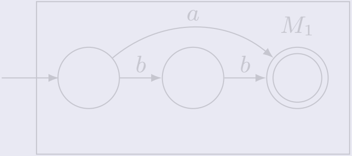
(b) M_2 accepts $L(ba^* + \lambda)$.

Example 3.7

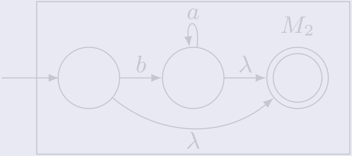
Find an NFA that accepts $L(r)$, where

$$r = (a + bb)^*(ba^* + \lambda).$$

Automata for $(a + bb)$ and $(ba^* + \lambda)$, constructed directly from first principles, are given in the following Figure.



(a) M_1 accepts $L(a + bb)$.



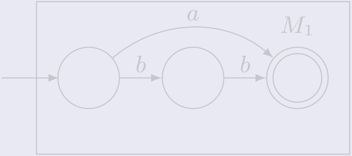
(b) M_2 accepts $L(ba^* + \lambda)$.

Example 3.7

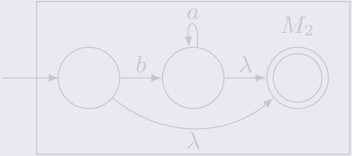
Find an NFA that accepts $L(r)$, where

$$r = (a + bb)^* (ba^* + \lambda).$$

Automata for $(a + bb)$ and $(ba^* + \lambda)$, constructed directly from first principles, are given in the following Figure.



(a) M_1 accepts $L(a + bb)$.



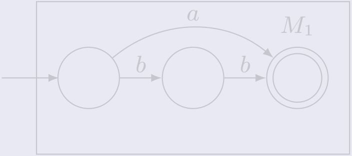
(b) M_2 accepts $L(ba^* + \lambda)$.

Example 3.7

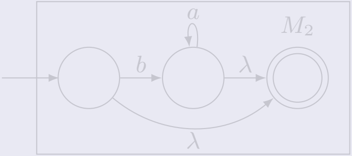
Find an NFA that accepts $L(r)$, where

$$r = (a + bb)^*(ba^* + \lambda).$$

Automata for $(a + bb)$ and $(ba^* + \lambda)$, constructed directly from first principles, are given in the following Figure.



(a) M_1 accepts $L(a + bb)$.



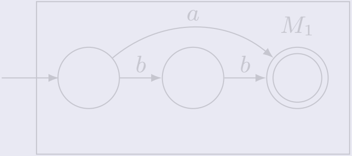
(b) M_2 accepts $L(ba^* + \lambda)$.

Example 3.7

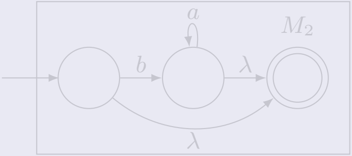
Find an NFA that accepts $L(r)$, where

$$r = (a + bb)^*(ba^* + \lambda).$$

Automata for $(a + bb)$ and $(ba^* + \lambda)$, constructed directly from first principles, are given in the following Figure.



(a) M_1 accepts $L(a + bb)$.



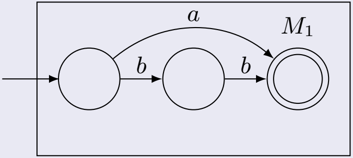
(b) M_2 accepts $L(ba^* + \lambda)$.

Example 3.7

Find an NFA that accepts $L(r)$, where

$$r = (a + bb)^*(ba^* + \lambda).$$

Automata for $(a + bb)$ and $(ba^* + \lambda)$, constructed directly from first principles, are given in the following Figure.



(a) M_1 accepts $L(a + bb)$.

(b) M_2 accepts $L(ba^* + \lambda)$.

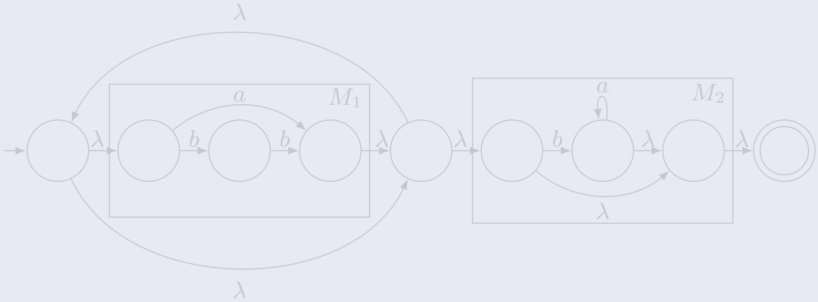
Example 3.7 (continuation)

Putting these together using the construction in Theorem 3.1, we get the solution in the Figure.

Automaton accepts $L((a + bb)^*(ba^* + \lambda))$.

Example 3.7 (continuation)

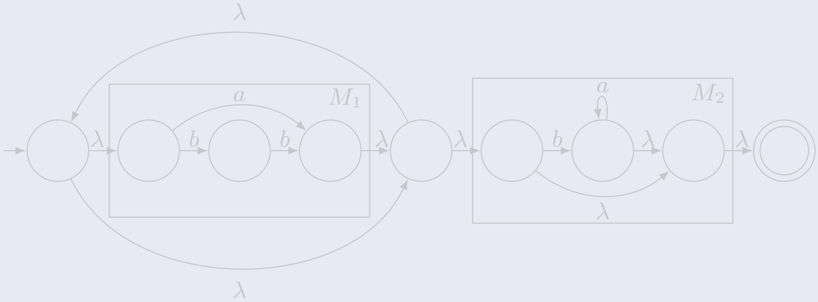
Putting these together using the construction in Theorem 3.1, we get the solution in the Figure.



Automaton accepts $L((a + bb)^*(ba^* + \lambda))$.

Example 3.7 (continuation)

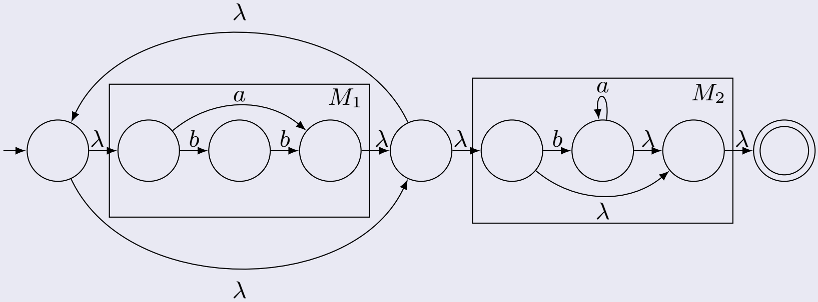
Putting these together using the construction in Theorem 3.1, we get the solution in the Figure.



Automaton accepts $L((a+bb)^*(ba^*+\lambda))$.

Example 3.7 (continuation)

Putting these together using the construction in Theorem 3.1, we get the solution in the Figure.



Automaton accepts $L((a+bb)^*(ba^*+\lambda))$.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully.

There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully.

There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully.

There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully.

There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

Regular Expressions for Regular Languages

It is intuitively reasonable that the converse of Theorem 3.1 should hold, and that for every regular language, there should exist a corresponding regular expression. Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state. This does not look too difficult but it is complicated by the existence of cycles that can often be traversed arbitrarily, in any order. This creates a bookkeeping problem that must be handled carefully. There are several ways to do this; one of the more intuitive approaches requires a side trip into what are called *generalized transition graphs* (*GTG*). Since this idea is used here in a limited way and plays no role in our further discussion, we shall deal with it informally.

A generalized transition graph is a transition graph whose edges are labeled with regular expressions; otherwise it is the same as the usual transition graph. The label of any walk from the initial state to a final state is the concatenation of several regular expressions, and hence itself a regular expression. The strings denoted by such regular expressions are a subset of the language accepted by the generalized transition graph, with the full language being the union of all such generated subsets.

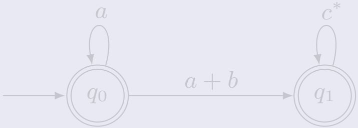
Example 3.8

The following Figure represents a generalized transition graph.

The language accepted by it is $L(a^* + a^*(a+b)c^*)$, as should be clear from an inspection of the graph. The edge (q_0, q_0) labeled a is a cycle that can generate any number of a's, that is, it represents $L(a^*)$. We could have labeled this edge a^* without changing the language accepted by the graph.

Example 3.8

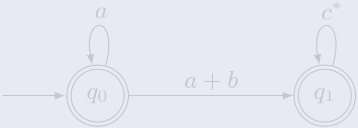
The following Figure represents a generalized transition graph.



The language accepted by it is $L(a^* + a^*(a+b)c^*)$, as should be clear from an inspection of the graph. The edge (q_0, q_0) labeled a is a cycle that can generate any number of a's, that is, it represents $L(a^*)$. We could have labeled this edge a^* without changing the language accepted by the graph.

Example 3.8

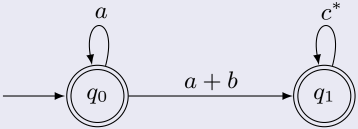
The following Figure represents a generalized transition graph.



The language accepted by it is $L(a^* + a^*(a+b)c^*)$, as should be clear from an inspection of the graph. The edge (q_0, q_0) labeled a is a cycle that can generate any number of a's, that is, it represents $L(a^*)$. We could have labeled this edge a^* without changing the language accepted by the graph.

Example 3.8

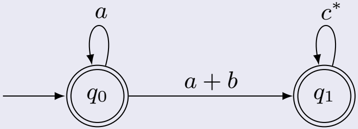
The following Figure represents a generalized transition graph.



The language accepted by it is $L(a^* + a^*(a+b)c^*)$, as should be clear from an inspection of the graph. The edge (q_0, q_0) labeled a is a cycle that can generate any number of a's, that is, it represents $L(a^*)$. We could have labeled this edge a^* without changing the language accepted by the graph.

Example 3.8

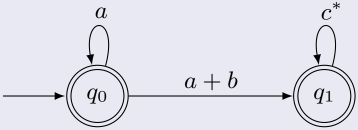
The following Figure represents a generalized transition graph.



The language accepted by it is $L(a^* + a^*(a+b)c^*)$, as should be clear from an inspection of the graph. The edge (q_0, q_0) labeled a is a cycle that can generate any number of a's, that is, it represents $L(a^*)$. We could have labeled this edge a^* without changing the language accepted by the graph.

Example 3.8

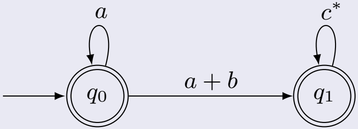
The following Figure represents a generalized transition graph.



The language accepted by it is $L(a^* + a^*(a+b)c^*)$, as should be clear from an inspection of the graph. The edge (q_0, q_0) labeled a is a cycle that can generate any number of a's, that is, it represents $L(a^*)$. We could have labeled this edge a^* without changing the language accepted by the graph.

Example 3.8

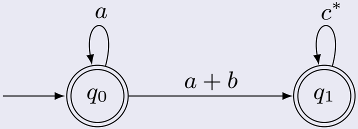
The following Figure represents a generalized transition graph.



The language accepted by it is $L(a^* + a^*(a+b)c^*)$, as should be clear from an inspection of the graph. The edge (q_0, q_0) labeled a is a cycle that can generate any number of a's, that is, it represents $L(a^*)$. We could have labeled this edge a^* without changing the language accepted by the graph.

Example 3.8

The following Figure represents a generalized transition graph.



The language accepted by it is $L(a^* + a^*(a+b)c^*)$, as should be clear from an inspection of the graph. The edge (q_0, q_0) labeled a is a cycle that can generate any number of a's, that is, it represents $L(a^*)$. We could have labeled this edge a^* without changing the language accepted by the graph.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

The graph of any nondeterministic finite accepter can be considered a generalized transition graph if the edge labels are interpreted properly. An edge labeled with a single symbol a is interpreted as an edge labeled with the expression a, while an edge labeled with multiple symbols $a, b, \ldots$ is interpreted as an edge labeled with the expression $a + b + \ldots$. From this observation, it follows that for every regular language, there exists a generalized transition graph that accepts it. Conversely, every language accepted by a generalized transition graph is regular. Since the label of every walk in a generalized transition graph is a regular expression, this appears to be an immediate consequence of Theorem 3.1.

Equivalence for generalized transition graphs is defined in terms of the language accepted and the purpose of the next bit of discussion is to produce a sequence of increasingly simple GTGs. In this, we will find it convenient to work with complete GTGs. A complete GTG is a graph in which all edges are present. If a GTG, after conversion from an NFA, has some edges missing, we put them in and label them with $\varnothing$. A complete GTG with $|V|$ vertices has exactly $|V|^2$ edges.

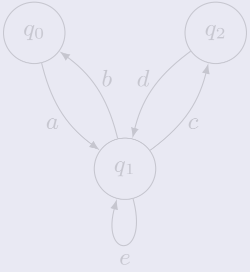
Example 3.9

The GTG in Figure (a) is not complete. Figure (b) shows how it is completed.

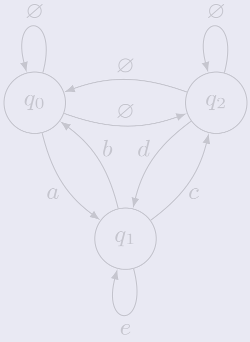
(a) (b)

Example 3.9

The GTG in Figure (a) is not complete. Figure (b) shows how it is completed.



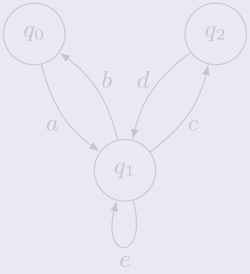
(a)



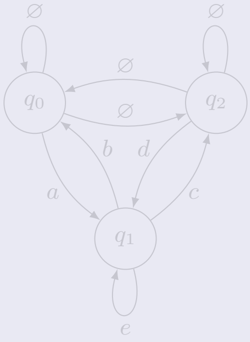
(b)

Example 3.9

The GTG in Figure (a) is not complete. Figure (b) shows how it is completed.



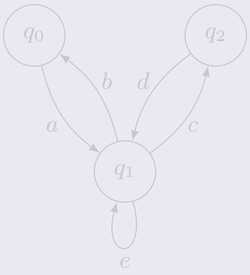
(a)



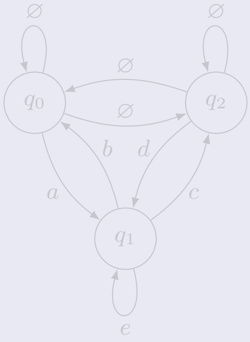
(b)

Example 3.9

The GTG in Figure (a) is not complete. Figure (b) shows how it is completed.



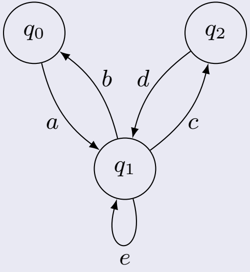
(a)



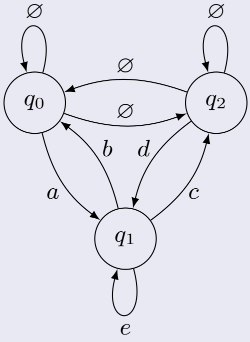
(b)

Example 3.9

The GTG in Figure (a) is not complete. Figure (b) shows how it is completed.



(a)



(b)

Suppose now that we have the simple two-state complete GTG shown in the following Figure.

By mentally tracing through this GTG you can convince yourself that the regular expression

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \tag{1}$$

covers all possible paths and so is the correct regular expression associated with the graph.

When a GTG has more than two states, we can find an equivalent graph by removing one state at a time. We shall illustrate this with an example before going to the general method.

Suppose now that we have the simple two-state complete GTG shown in the following Figure.

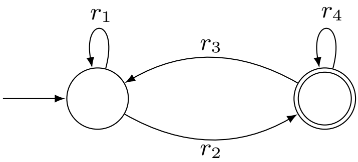
By mentally tracing through this GTG you can convince yourself that the regular expression

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \tag{1}$$

covers all possible paths and so is the correct regular expression associated with the graph.

When a GTG has more than two states, we can find an equivalent graph by removing one state at a time. We shall illustrate this with an example before going to the general method.

Suppose now that we have the simple two-state complete GTG shown in the following Figure.



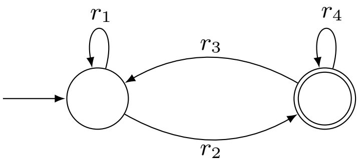
By mentally tracing through this GTG you can convince yourself that the regular expression

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \tag{1}$$

covers all possible paths and so is the correct regular expression associated with the graph.

When a GTG has more than two states, we can find an equivalent graph by removing one state at a time. We shall illustrate this with an example before going to the general method.

Suppose now that we have the simple two-state complete GTG shown in the following Figure.



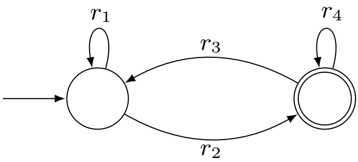
By mentally tracing through this GTG you can convince yourself that the regular expression

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \quad (1)$$

covers all possible paths and so is the correct regular expression associated with the graph.

When a GTG has more than two states, we can find an equivalent graph by removing one state at a time. We shall illustrate this with an example before going to the general method.

Suppose now that we have the simple two-state complete GTG shown in the following Figure.



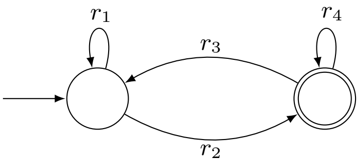
By mentally tracing through this GTG you can convince yourself that the regular expression

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \tag{1}$$

covers all possible paths and so is the correct regular expression associated with the graph.

When a GTG has more than two states, we can find an equivalent graph by removing one state at a time. We shall illustrate this with an example before going to the general method.

Suppose now that we have the simple two-state complete GTG shown in the following Figure.



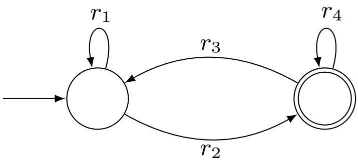
By mentally tracing through this GTG you can convince yourself that the regular expression

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \tag{1}$$

covers all possible paths and so is the correct regular expression associated with the graph.

When a GTG has more than two states, we can find an equivalent graph by removing one state at a time. We shall illustrate this with an example before going to the general method.

Suppose now that we have the simple two-state complete GTG shown in the following Figure.



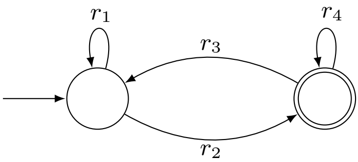
By mentally tracing through this GTG you can convince yourself that the regular expression

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \tag{1}$$

covers all possible paths and so is the correct regular expression associated with the graph.

When a GTG has more than two states, we can find an equivalent graph by removing one state at a time. We shall illustrate this with an example before going to the general method.

Suppose now that we have the simple two-state complete GTG shown in the following Figure.



By mentally tracing through this GTG you can convince yourself that the regular expression

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \quad (1)$$

covers all possible paths and so is the correct regular expression associated with the graph.

When a GTG has more than two states, we can find an equivalent graph by removing one state at a time. We shall illustrate this with an example before going to the general method.

Example 3.10

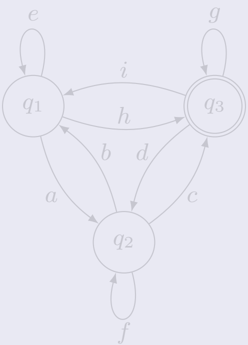
Consider the complete GTG in the following Figure.

To remove q_2, we first introduce some new edges. We

- create an edge from q_1 to q_1 and label it $e + af^*b$,
- create an edge from q_1 to q_3 and label it $h + af^*c$,
- create an edge from q_3 to q_1 and label it $i + df^*b$,
- create an edge from q_3 to q_3 and label it $g + df^*c$.

Example 3.10

Consider the complete GTG in the following Figure.

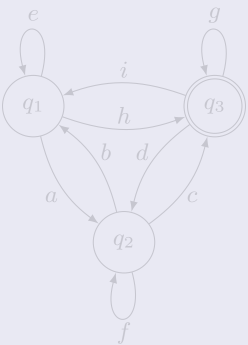


To remove q_2, we first introduce some new edges. We

- create an edge from q_1 to q_1 and label it $e + af^*b$,
- create an edge from q_1 to q_3 and label it $h + af^*c$,
- create an edge from q_3 to q_1 and label it $i + df^*b$,
- create an edge from q_3 to q_3 and label it $g + df^*c$.

Example 3.10

Consider the complete GTG in the following Figure.

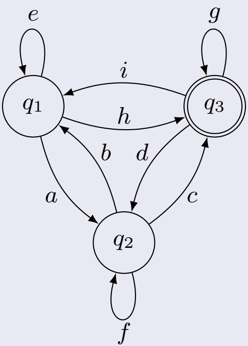


To remove q_2, we first introduce some new edges. We

- create an edge from q_1 to q_1 and label it $e + af^*b$,
- create an edge from q_1 to q_3 and label it $h + af^*c$,
- create an edge from q_3 to q_1 and label it $i + df^*b$,
- create an edge from q_3 to q_3 and label it $g + df^*c$.

Example 3.10

Consider the complete GTG in the following Figure.

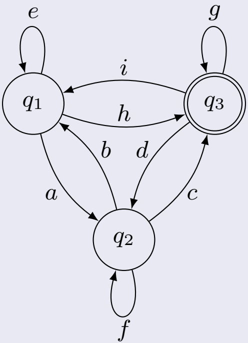


To remove q_2, we first introduce some new edges. We

- create an edge from q_1 to q_1 and label it $e + af^*b$,
- create an edge from q_1 to q_3 and label it $h + af^*c$,
- create an edge from q_3 to q_1 and label it $i + df^*b$,
- create an edge from q_3 to q_3 and label it $g + df^*c$.

Example 3.10

Consider the complete GTG in the following Figure.

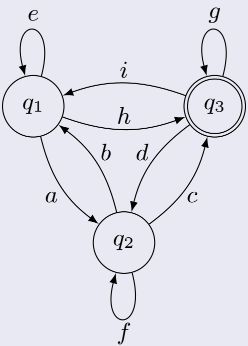


To remove q_2, we first introduce some new edges. We

- create an edge from q_1 to q_1 and label it $e + af^*b$,
- create an edge from q_1 to q_3 and label it $h + af^*c$,
- create an edge from q_3 to q_1 and label it $i + df^*b$,
- create an edge from q_3 to q_3 and label it $g + df^*c$.

Example 3.10

Consider the complete GTG in the following Figure.

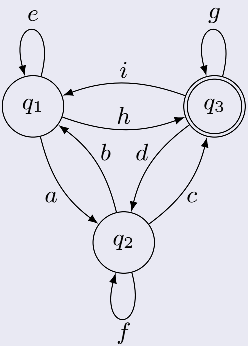


To remove q_2, we first introduce some new edges. We

- create an edge from q_1 to q_1 and label it $e + af^*b$,
- create an edge from q_1 to q_3 and label it $h + af^*c$,
- create an edge from q_3 to q_1 and label it $i + df^*b$,
- create an edge from q_3 to q_3 and label it $g + df^*c$.

Example 3.10

Consider the complete GTG in the following Figure.



To remove q_2, we first introduce some new edges. We

create an edge from q_1 to q_1 and label it $e + af^*b$,

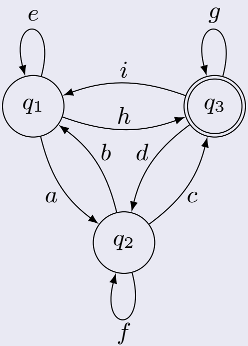
create an edge from q_1 to q_3 and label it $h + af^*c$,

create an edge from q_3 to q_1 and label it $i + df^*b$,

create an edge from q_3 to q_3 and label it $g + df^*c$.

Example 3.10

Consider the complete GTG in the following Figure.



To remove q_2, we first introduce some new edges. We

create an edge from q_1 to q_1 and label it $e + af^*b$,

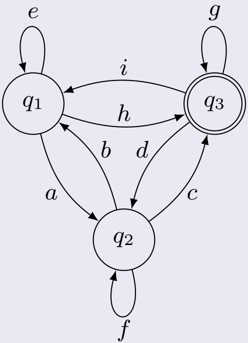
create an edge from q_1 to q_3 and label it $h + af^*c$,

create an edge from q_3 to q_1 and label it $i + df^*b$,

create an edge from q_3 to q_3 and label it $g + df^*c$.

Example 3.10

Consider the complete GTG in the following Figure.



To remove q_2, we first introduce some new edges. We

create an edge from q_1 to q_1 and label it $e + af^*b$,

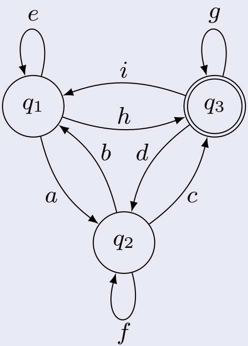
create an edge from q_1 to q_3 and label it $h + af^*c$,

create an edge from q_3 to q_1 and label it $i + df^*b$,

create an edge from q_3 to q_3 and label it $g + df^*c$.

Example 3.10

Consider the complete GTG in the following Figure.



To remove q_2, we first introduce some new edges. We

create an edge from q_1 to q_1 and label it $e + af^*b$,

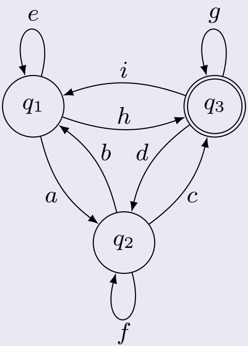
create an edge from q_1 to q_3 and label it $h + af^*c$,

create an edge from q_3 to q_1 and label it $i + df^*b$,

create an edge from q_3 to q_3 and label it $g + df^*c$.

Example 3.10

Consider the complete GTG in the following Figure.



To remove q_2, we first introduce some new edges. We

create an edge from q_1 to q_1 and label it $e + af^*b$,

create an edge from q_1 to q_3 and label it $h + af^*c$,

create an edge from q_3 to q_1 and label it $i + df^*b$,

create an edge from q_3 to q_3 and label it $g + df^*c$.

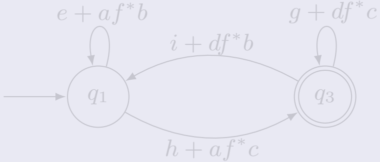
Example 3.10

When this is done, we remove q_2 and all associated edges. This gives the GTG in the following Figure.

You can explore the equivalence of the two GTGs by seeing how regular expressions such as af^*c and e^*ab are generated.

Example 3.10

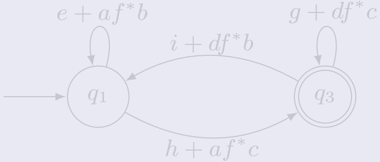
When this is done, we remove q_2 and all associated edges. This gives the GTG in the following Figure.



You can explore the equivalence of the two GTGs by seeing how regular expressions such as af^*c and e^*ab are generated.

Example 3.10

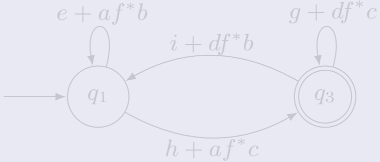
When this is done, we remove q_2 and all associated edges. This gives the GTG in the following Figure.



You can explore the equivalence of the two GTGs by seeing how regular expressions such as af^*c and e^*ab are generated.

Example 3.10

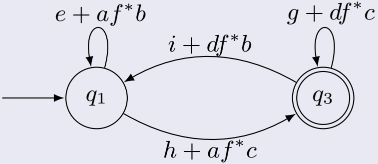
When this is done, we remove q_2 and all associated edges. This gives the GTG in the following Figure.



You can explore the equivalence of the two GTGs by seeing how regular expressions such as af^*c and e^*ab are generated.

Example 3.10

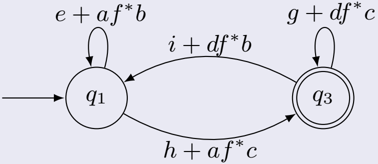
When this is done, we remove q_2 and all associated edges. This gives the GTG in the following Figure.



You can explore the equivalence of the two GTGs by seeing how regular expressions such as af^*c and e^*ab are generated.

Example 3.10

When this is done, we remove q_2 and all associated edges. This gives the GTG in the following Figure.



You can explore the equivalence of the two GTGs by seeing how regular expressions such as af^*c and e^*ab are generated.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \qquad (2)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \qquad (3)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^* \qquad (2)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \qquad (3)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \emptyset = r$, $r\emptyset = \emptyset$, $\emptyset^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

3.2 Regular Expressions and Regular Languages

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^* \quad (2)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \quad (3)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \emptyset = r$, $r\emptyset = \emptyset$, $\emptyset^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^* \qquad (1)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \qquad (2)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \qquad (2)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \qquad (3)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \emptyset = r$, $r\emptyset = \emptyset$, $\emptyset^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \quad (2)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \quad (3)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^* . \qquad (2)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \qquad (3)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij}(r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \quad (2)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \quad (3)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \qquad (2)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \qquad (3)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \qquad (2)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \qquad (3)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij}(r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \quad (2)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \quad (3)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \tag{2}$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \tag{3}$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

For arbitrary GTGs we remove one state at a time until only two states are left. Then we apply Equation (1) to get the final regular expression. This tends to be a lengthy process, but it is straightforward as the following procedure shows.

Procedure: NFA-to-rex

1. Start with an NFA with states $q_0, q_1, \ldots, q_n$, and a single final state, distinct from its initial state.
2. Convert the NFA into a complete generalized transition graph. Let r_{ij} stand for the label of the edge from q_i to q_j.
3. If the GTG has only two states, with q_i as its initial state and q_j its final state, its associated regular expression is
$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*. \qquad (2)$$
4. If the GTG has three states, with initial state q_i, final state q_j, and third state q_k, introduce new edges, labeled
$$r_{pq} + r_{pk} r_{kk}^* r_{kq} \qquad (3)$$
for $p = i, j$, $q = i, j$. When this is done, remove vertex q_k and its associated edges.
5. If the GTG has four or more states, pick a state q_k to be removed. Apply rule 4 for all pairs of states (q_i, q_j), $i \neq k$, $j \neq k$. At each step apply the simplifying rules $r + \varnothing = r$, $r\varnothing = \varnothing$, $\varnothing^* = \lambda$, wherever possible. When this is done, remove state q_k.
6. Repeat Steps 3 to 5 until the correct regular expression is obtained.

Example 3.11

Find a regular expression for the language

$$L = \{w \in \{a, b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd}\}.$$

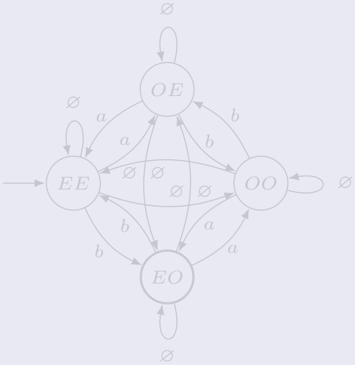
An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an NFA for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in the following Figure

Example 3.11

Find a regular expression for the language

$$L = \left\{w \in \{a, b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd}\right\}.$$

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an NFA for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in the following Figure

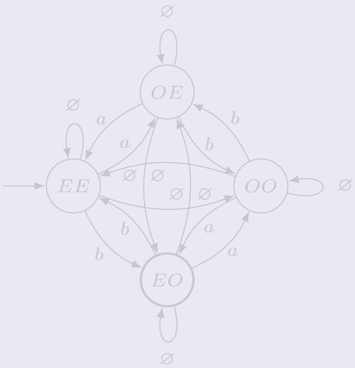


Example 3.11

Find a regular expression for the language

$$L = \left\{ w \in \{a, b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd} \right\}.$$

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an NFA for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in the following Figure

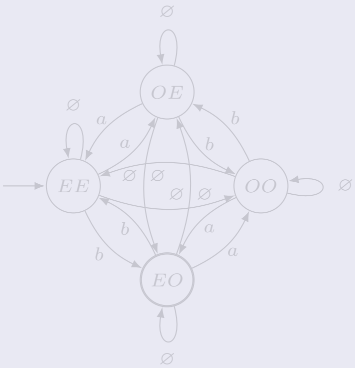


Example 3.11

Find a regular expression for the language

$$L = \left\{w \in \{a, b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd}\right\}.$$

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an NFA for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in the following Figure

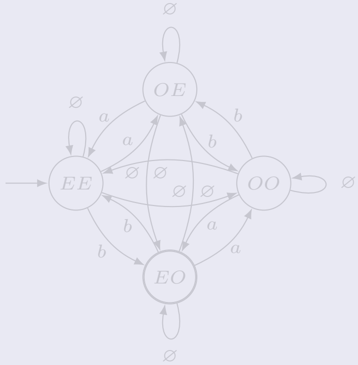


Example 3.11

Find a regular expression for the language

$$L = \left\{w \in \{a, b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd}\right\}.$$

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an NFA for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in the following Figure

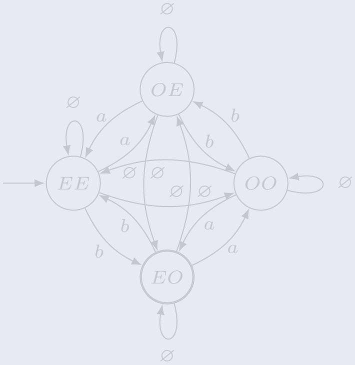


Example 3.11

Find a regular expression for the language

$$L = \left\{ w \in \{a, b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd} \right\}.$$

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an NFA for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in the following Figure

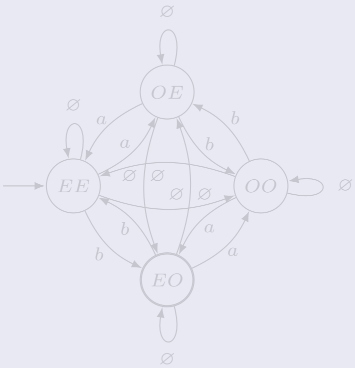


Example 3.11

Find a regular expression for the language

$$L = \{w \in \{a, b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd}\}.$$

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an NFA for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in the following Figure

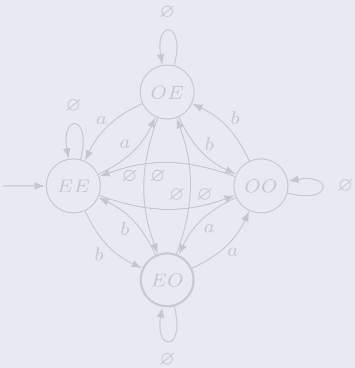


Example 3.11

Find a regular expression for the language

$$L = \left\{ w \in \{a, b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd} \right\}.$$

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an NFA for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in the following Figure

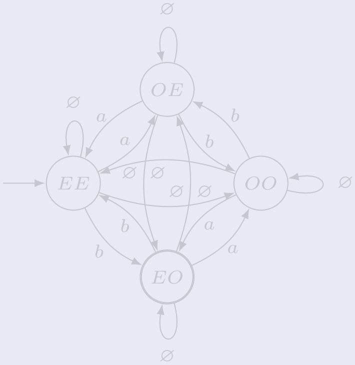


Example 3.11

Find a regular expression for the language

$$L = \{w \in \{a,b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd}\}.$$

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an NFA for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in the following Figure

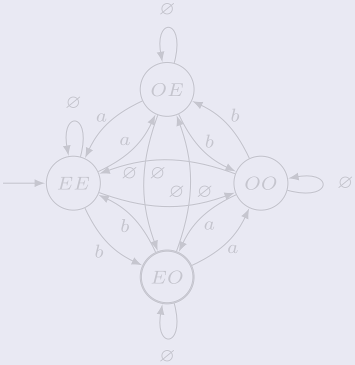


Example 3.11

Find a regular expression for the language

$$L = \left\{ w \in \{a, b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd} \right\}.$$

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an NFA for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in the following Figure

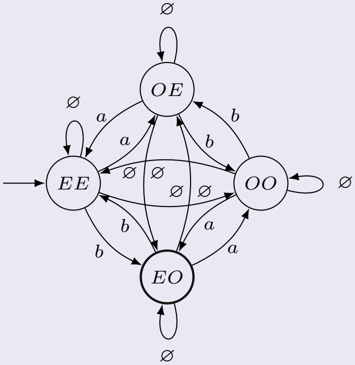


Example 3.11

Find a regular expression for the language

$$L = \{w \in \{a, b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd}\}.$$

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an NFA for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in the following Figure



Example 3.11 (continuation)

We now apply the conversion to a regular expression, using procedure NFA-to-rex. To remove the state OE, we apply Equation (3). The edge between EE and itself will have the label

$$\begin{aligned} r_{EE} &= \varnothing + a\varnothing^{*}a = \\ &= aa \end{aligned}$$

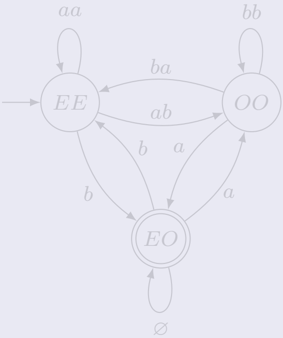
We continue in this manner until we get the GTG in the Figure.

Example 3.11 (continuation)

We now apply the conversion to a regular expression, using procedure NFA-to-rex. To remove the state OE, we apply Equation (3). The edge between EE and itself will have the label

$$r_{EE} = \varnothing + a\varnothing^* a = \\ = aa$$

We continue in this manner until we get the GTG in the Figure.

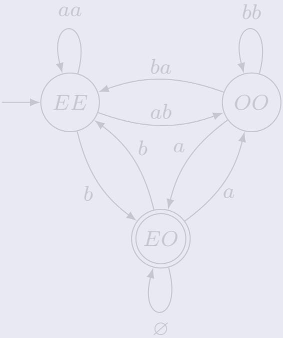


Example 3.11 (continuation)

We now apply the conversion to a regular expression, using procedure NFA-to-rex. To remove the state OE, we apply Equation (3). The edge between EE and itself will have the label

$$r_{EE} = \varnothing + a\varnothing^* a =$$
$$= aa$$

We continue in this manner until we get the GTG in the Figure.

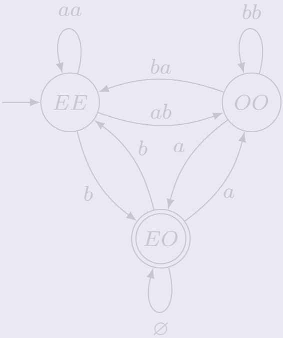


Example 3.11 (continuation)

We now apply the conversion to a regular expression, using procedure NFA-to-rex. To remove the state OE, we apply Equation (3). The edge between EE and itself will have the label

$$r_{EE} = \varnothing + a\varnothing^* a = \\ = aa$$

We continue in this manner until we get the GTG in the Figure.

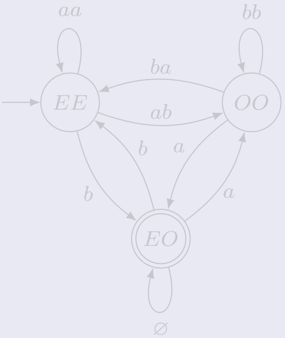


Example 3.11 (continuation)

We now apply the conversion to a regular expression, using procedure NFA-to-rex. To remove the state OE, we apply Equation (3). The edge between EE and itself will have the label

$$r_{EE} = \varnothing + a\varnothing^{*}a =$$
$$= aa$$

We continue in this manner until we get the GTG in the Figure.

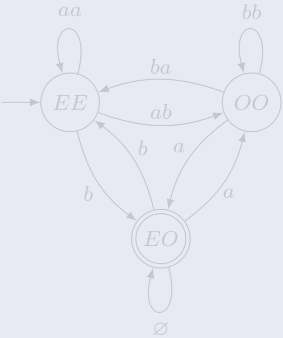


Example 3.11 (continuation)

We now apply the conversion to a regular expression, using procedure NFA-to-rex. To remove the state OE, we apply Equation (3). The edge between EE and itself will have the label

$$r_{EE} = \varnothing + a\varnothing^* a =$$
$$= aa$$

We continue in this manner until we get the GTG in the Figure.

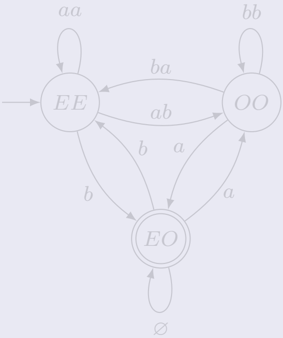


Example 3.11 (continuation)

We now apply the conversion to a regular expression, using procedure NFA-to-rex. To remove the state OE, we apply Equation (3). The edge between EE and itself will have the label

$$\begin{aligned} r_{EE} &= \varnothing + a\varnothing^* a = \\ &= aa \end{aligned}$$

We continue in this manner until we get the GTG in the Figure.

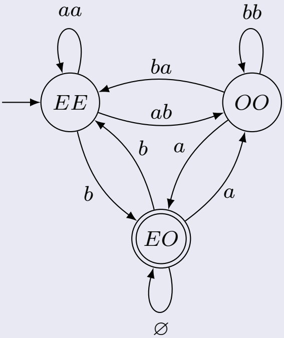


Example 3.11 (continuation)

We now apply the conversion to a regular expression, using procedure NFA-to-rex. To remove the state OE, we apply Equation (3). The edge between EE and itself will have the label

$$r_{EE} = \varnothing + a\varnothing^* a = \\ = aa$$

We continue in this manner until we get the GTG in the Figure.



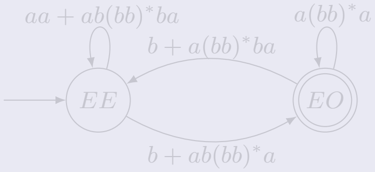
Example 3.11 (continuation)

We continue in this manner until we get the GTG in the Figure.

Finally, we get the correct regular expression from Equation (2).

Example 3.11 (continuation)

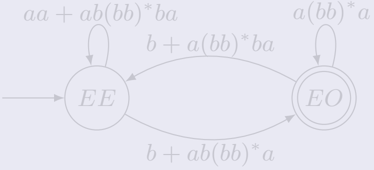
We continue in this manner until we get the GTG in the Figure.



Finally, we get the correct regular expression from Equation (2).

Example 3.11 (continuation)

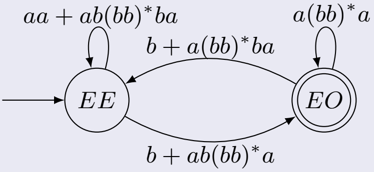
We continue in this manner until we get the GTG in the Figure.



Finally, we get the correct regular expression from Equation (2).

Example 3.11 (continuation)

We continue in this manner until we get the GTG in the Figure.



Finally, we get the correct regular expression from Equation (2).

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ■

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r.

While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ■

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ■

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

The process of converting an NFA to a regular expression is mechanical but tedious. It leads to regular expressions that are complicated and of little practical use. The main reason for presenting this process is that it gives the idea for the proof of an important result.

Theorem 3.2

Let L be a regular language. Then there exists a regular expression r such that $L = L(r)$.

Proof. If L is regular, there exists an NFA for it. We can assume without loss of generality that this NFA has a single final state, distinct from its initial state. We convert this NFA to a complete generalized transition graph and apply the procedure NFA-to-rex to it. This yields the required regular expression r. While this can make the result plausible, a rigorous proof requires that we show that each step in the process generates an equivalent GTG. This is a technical matter we leave to the reader. ∎

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Regular Expressions for Describing Simple Patterns

In previous lectures we explored the connection between finite accepters and some of the simpler constituents of program ming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12

An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the vi editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

Example 3.12 (continuation)

If the pattern is specified by a regular expression, the pattern recognition program can take this description and convert it into an equivalent NFA using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a DFA. This DFA, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must also be considered. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Lecture 7 is helpful.

Example 3.12 (continuation)

If the pattern is specified by a regular expression, the pattern recognition program can take this description and convert it into an equivalent NFA using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a DFA. This DFA, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must also be considered. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Lecture 7 is helpful.

Example 3.12 (continuation)

If the pattern is specified by a regular expression, the pattern recognition program can take this description and convert it into an equivalent NFA using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a DFA. This DFA, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must also be considered. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Lecture 7 is helpful.

Example 3.12 (continuation)

If the pattern is specified by a regular expression, the pattern recognition program can take this description and convert it into an equivalent NFA using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a DFA. This DFA, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must also be considered. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Lecture 7 is helpful.

Example 3.12 (continuation)

If the pattern is specified by a regular expression, the pattern recognition program can take this description and convert it into an equivalent NFA using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a DFA. This DFA, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must also be considered. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Lecture 7 is helpful.

Example 3.12 (continuation)

If the pattern is specified by a regular expression, the pattern recognition program can take this description and convert it into an equivalent NFA using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a DFA. This DFA, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must also be considered. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Lecture 7 is helpful.

Example 3.12 (continuation)

If the pattern is specified by a regular expression, the pattern recognition program can take this description and convert it into an equivalent NFA using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a DFA. This DFA, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must also be considered. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Lecture 7 is helpful.

Example 3.12 (continuation)

If the pattern is specified by a regular expression, the pattern recognition program can take this description and convert it into an equivalent NFA using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a DFA. This DFA, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must also be considered. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Lecture 7 is helpful.

Example 3.12 (continuation)

If the pattern is specified by a regular expression, the pattern recognition program can take this description and convert it into an equivalent NFA using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a DFA. This DFA, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must also be considered. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Lecture 7 is helpful.

Example 3.12 (continuation)

If the pattern is specified by a regular expression, the pattern recognition program can take this description and convert it into an equivalent NFA using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a DFA. This DFA, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must also be considered. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Lecture 7 is helpful.

Thank You for attention!