Formal Languages, Automata and Codes

Oleg Gutik

Lecture 12

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ■

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

---

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

4.2 Elementary Questions about Regular Languages

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

---

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

---

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

---

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ■

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ■

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ■

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

4.2 Elementary Questions about Regular Languages

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ■

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ■

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ■

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ■

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ■

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ∎

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

4.2 Elementary Questions about Regular Languages

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ■

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

4.2 Elementary Questions about Regular Languages

We now come to a very fundamental issue: Given a language L and a string w, can we determine whether or not w is an element of L? This is the **membership** question and a method for answering it is called a membership algorithm.[1] Very little can be done with languages for which we cannot find efficient membership algorithms. The question of the existence and nature of membership algorithms will be of great concern in later discussions; it is an issue that is often difficult. For regular languages, though, it is an easy matter.

We first consider what exactly we mean when we say "given a language ...". In many arguments, it is important that this be unambiguous. We have used several ways of describing regular languages: informal verbal descriptions, set notation, finite automata, regular expressions, and regular grammars. Only the last three are sufficiently well defined for use in theorems. We therefore say that a regular language is given in a **standard representation** if and only if it is described by a finite automaton, a regular expression, or a regular grammar.

Theorem 4.5

Given a standard representation of any regular language L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.

Proof. We represent the language by some DFA, then test w to see if it is accepted by this automaton. ■

[1] Later we shall make precise what the term "algorithm" means. For the moment, think of it as a method for which one can write a computer program.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

4.2 Elementary Questions about Regular Languages

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

4.2 Elementary Questions about Regular Languages

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

4.2 Elementary Questions about Regular Languages

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

4.2 Elementary Questions about Regular Languages

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ∎

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Other important questions are whether a language is finite or infinite, whether two languages are the same, and whether one language is a subset of another. For regular languages at least, these questions are easily answered.

Theorem 4.6

There exists an algorithm for determining whether a regular language, given in standard representation, is empty, finite, or infinite.

Proof. The answer is apparent if we represent the language as a transition graph of a DFA. If there is a simple path from the initial vertex to any final vertex, then the language is not empty.

To determine whether or not a language is infinite, find all the vertices that are the base of some cycle. If any of these are on a path from an initial to a final vertex, the language is infinite. Otherwise, it is finite. ■

The question of the equality of two languages is also an important practical issue. Often several definitions of a programming language exist, and we need to know whether, in spite of their different appearances, they specify the same language. This is generally a difficult problem; even for regular languages the argument is not obvious. It is not possible to argue on a sentence-by-sentence comparison, because this works only for finite languages. Nor is it easy to see the answer by looking at the regular expressions, grammars, or DFA's. An elegant solution uses the already established closure properties.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ■

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ∎

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ■

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ∎

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ■

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ∎

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ∎

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ■

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ∎

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ■

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ∎

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ■

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ∎

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ■

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ∎

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ∎

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ■

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Theorem 4.7

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$.

Proof. Using L_1 and L_2 we define the language

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

By closure, L_3 is regular, and we can find a DFA M that accepts L_3. Once we have M we can then use the algorithm in Theorem 4.6 to determine if L_3 is empty. But that $L_3 = \varnothing$ if and only if $L_1 = L_2$. ■

These results are fundamental, in spite of being obvious and unsurprising. For regular languages, the questions raised by Theorems 4.5 to 4.7 can be answered easily, but this is not always the case when we deal with other language families. We shall encounter questions like these on several occasions later on. Anticipating a little, we shall see that the answers become increasingly more difficult and eventually impossible to find.

Thank You for attention!