Welcome!

INTRODUCTION TO COMPUTATION THEORY

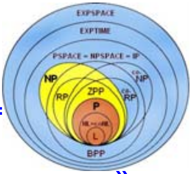
Jinkui Xie

jkxie@cs.ecnu.edu.cn

Slides for a Course Based on the Text

Introduction to Automata Theory, Languages, and Computation (2nd Edition)

BY JOHN E. HOPCROFT, RAJEEV MOTWANI, JEFFREY D. ULLMAN



East China Normal University

KEY COURSE PROJECT FOR POSTGRADUATE

INTRODUCTION TO COMPUTATION THEORY

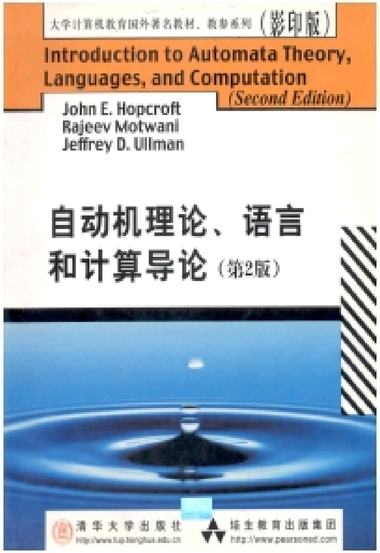
Jinkui Xie
jkxie@cs.ecnu.edu.cn

Slides for a Course Based on the Text

Introduction to Automata Theory, Languages, and Computation (2nd Edition)

BY JOHN E. HOPCROFT, RAJEEV MOTWANI, JEFFREY D. ULLMAN

TEXTBOOK

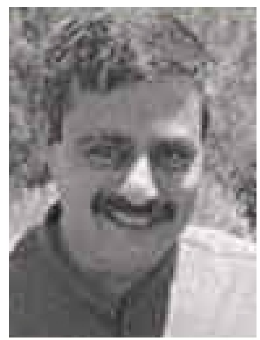


English Version

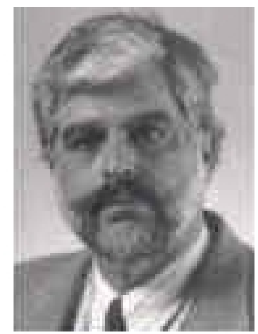
Chinese Version

AUTHORS

J.E.Hopcroft



R.Motwani



J.D.Ullman

COURSE SUBJECT

- This course is an introduction to the **Theory of Computation**.
- Sounds impressive!
- But what does **Theory of Computation** mean?

THEORY

- Study of **abstractions** of reality.
- Irrelevant complications are dropped in order to isolate important concepts.
- Studying the "theory of subject X" means that simplified versions of X are analyzed from various perspectives.
- So the objects being analyzed in a theory are supposed to be **simple**.
- Doesn't mean that the subject is easy!

COMPUTATION

- There's more than one kind of computation.
- Our general approach is to start with very simple, deliberately restricted models of computers.
- Goal is to understand them and then to proceed more complex models.
- We will study three models: Automata, Grammars, and Turing Machines.
- We work with each model in order to see what can be done with it.
- We'll reason about the models, e.g. to prove the relationship between them.

WHY STUDY THEORY?

A questions commonly posed by practised minded students! Here is some answer.

- Theory gives exposure to *ideas* that permeate Computer Science: logic, sets, recursion, automata, grammars. Familiar with these concepts will make you a better computer scientist.
- Theory gives us mathematical (hence precise) descriptions of computational phenomena. This allow us to use mathematics to solve problems arising from computers.
- It gives training in argumentation, which is generally useful thing.

WHY STUDY THEORY?

A questions commonly posed by practised minded students! Here is some answer.

- It is required if you are interested in a research career in Computer Science.
- A theory course distinguishes you from someone who has picked up programming at a “job factory” technique school.
- Theory gives exposure to some of the absolute highpoints of human thought.
- Theory gives a nice setting for honing your problem solving skills.
- You will get much practice in solving problems in this course.

SYLLABI

- Preliminaries — 1 SESSION
- Automata — 5 SESSIONS

  (Finite Automata, Regular Expressions, Regular Grammars, Properties of Regular Languages)
- Grammars — 5 SESSIONS

  (Context-Free Grammars and Languages, Pushdown Automata, Properties of Context-Free Languages)
- Turing Machines — 5 SESSIONS

  (Introduction to Turing Machines, Computability, Complexity)

PREREQUISITES

- You will need a good working knowledge of the material for this course.
- It is highly recommended that you brush up on the following topics: sets, relations, functions, logic, proof techniques, graph theory, counting techniques, permutations and combinations, etc.

EXAMINATION

- Every students is required to submit one tractates (30% of total marks) for developing the course.
- The final examination (70% of total marks) will be a three-hour examination at the end of the term.
- To pass the course you must have a passing mark on the final examination and the tractates.

Let's begin our lesson ……

Session 1

- Inductive Proofs
- The Central Concepts of Automata Theory

Inductive Proofs

- Induction is an extremely important proof technique that can be used to prove assertions, it is used extensively to prove results about the a large variety of recursively defined objects.

- Many of the most familiar inductive proofs deal with integers, but in automate theory, we also need inductive proofs about recursively defined concept as expressions of various sorts.

Induction on Integers

In order to prove a statement $S(n)$, about an integer n. One common approach is to prove two things:

1. The *basis step*, where we show $S(i)$ for a particular integer i. (Usually, $i = 0$ or $i = 1$.)

2. The *inductive step*, where we assume $n \geq i$, where i is the basis integer, and we show that "if $S(n)$ then $S(n+1)$."

Intuitively, these two parts should convince us that $S(n)$ is true for all $n \geq i$.

Why Induction Is Valid?

The reason comes from the well-ordering property.

The Well-Ordering Property

Every nonempty set of nonnegative integers has a least element.

Proof Suppose $S(n)$ were false for one or more of those integers. Then, by the well-ordering property, there would have to be a smallest value of n, say j, for which $S(j)$ is false, and yet $j \geq i$.

Now j could not be i, because we prove in the basis part that $S(i)$ is true. Thus, j must be greater than i. We now know that $j - 1 \geq i$, and $S(j-1)$ is true.

However, we proved in the inductive part that if $n \geq i$, then $S(n)$ implies $S(n+1)$. Suppose $n = j - 1$. Then we know from the inductive step that $S(j-1)$ implies $S(j)$. Since we also know $S(j-1)$, we can conclude $S(j)$.

We have assumed the negation of which we wanted to prove; that is, assumed $S(j)$ was false for some $j \geq i$. In each case, we derived a contradiction, so $S(n)$ is true for all $n \geq i$. ◀

Thus, we proved our logical reasoning system:

The Induction Principle

If we prove $S(i)$ *and we prove that* $n \geq i$, $S(n)$ *implies* $S(n+1)$, *then we may conclude* $S(n)$ *for all* $n \geq i$.

Example The harmonic numbers H_k, $k = 1, 2, \cdots$ are defined by

$$H_k = 1 + \frac{1}{2} + \cdots + \frac{1}{k}.$$

Use induction principle to prove that

$$H_{2^n} \geq 1 + \frac{n}{2}.$$

This inequality can be used to show that the harmonic series is a divergent infinite series.

Proof Let $S(n)$ be the proposition that $H_{2^n} \geq 1 + n/2$. We need to prove that $S(n)$ is true for $n = 0, 1, 2, \cdots$.

Basis step: $S(0)$ is true, since $H_{2^0} = H_1 = 1 \geq 1 + 0/2$.

Inductive step: Assume that $S(n)$ is true, so that $H_{2^n} \geq 1 + n/2$. It must be shown that $S(n+1)$, which states that $H_{2^{n+1}} \geq 1 + (n+1)/2$, must also be true under this assumption.

This can be done since

$$
\begin{aligned}
H_{2^{n+1}} &= 1 + \cdots + \frac{1}{2^n} + \frac{1}{2^n + 1} + \cdots + \frac{1}{2^{n+1}} \\
&= H_{2^n} + \frac{1}{2^n + 1} + \cdots + \frac{1}{2^{n+1}} \\
&\geq \left(1 + \frac{n}{2}\right) + 2^n \cdot \frac{1}{2^{n+1}} \\
&\geq 1 + \frac{n+1}{2}.
\end{aligned}
$$

This establishes the inductive step of the proof. Thus, the inequality for the harmonic numbers is valid for all nonnegative integers n. ◀

Question　What is wrong with the following "proof" that

all horses are the same color?

Let $S(n)$ be the proposition that all the horses in a set of n horses are the same color.

Basis step: $S(1)$ is true.

Inductive step: Assume that $S(n)$ is true, so that all the horses in any set of n horses are the same color. Consider any $n+1$ horses; number these horses $1, 2, \cdots, n, n+1$.

Now the first n of these horses all must have the same color, and the last n of these must have the same color.

Since the set of the first n horses and the set of the last n horses overlap, all $n+1$ must be the same color. This shows that $S(n+1)$ is true and finishes the proof by induction. ◀

Sometimes an induction proof is made possible only by using a more general scheme than the one proposed. Two important generalizations of this scheme are:

1. Use several basis cases. i.e., we prove $S(i), S(i+1), \cdots, S(j)$ for some $j \geq i$.

2. In proving $S(n+1)$, $(n \geq j)$ use the truth of all the statements $S(i), S(i+1), \cdots, S(n)$ rather than just using $S(n)$.

The conclusion to be made from this basis and inductive step is that $S(n)$ is true for all $n \geq i$.

Example Show that if n is an integer grater than 1, the n can be written as the product of primes.

Proof Let $S(n)$ be the proposition that n can be written as the product of primes.

Basis step: $S(2)$ is true, since 2 can be written as the product of one prime, itself.

Inductive step: Assume that $S(k)$ is true for all positive integers k with $2 \leq k \leq n$. To complete the inductive step, it must shown that $S(n+1)$ is true under this assumption.

There are two cases to consider, namely, when $n + 1$ is prime and when $n + 1$ is composite. If $n + 1$ is prime, we immediately see that $S(n + 1)$ is true. Otherwise, $n + 1$ is composite and can be written as the product of two positive integers a and b with $2 \leq a \leq b < n + 1$.

By the induction hypothesis, both a and b can be written as the products of primes. Thus, if $n+1$ is composite, it can be written as the product of primes, namely, those primes in the factorization of a and those in factorization of b. ◀

Structural Inductions

In computer science theory, there are several recursively defined structures about which we need to prove statements. The familiar notions of trees and expressions are important examples.

Like inductions, all recursive definitions have a basis case, where one or more elementary structures are defined, and an inductive step, where more complex structures are defined in terms of previously defined structures.

Example The recursive definition of a tree:

Basis step: A single node is a tree, and that node is the root of the tree.

Inductive step: If $T_1, T_2, \cdots, T_k$ are trees, then we can form a new tree as follows:

1. Begin with a new node N, which is the root of the tree.
2. Add copies of all the trees $T_1, T_2, \cdots, T_k$.
3. Add edges from node N to the roots of each of the trees $T_1, T_2, \cdots, T_k$.

◀

Example Define expressions using the arithmetic operators + and $*$, with both numbers and variables allowed as operands.

Basis step: Any number or letter is an expression.

Inductive step: If E and F are expressions, the so are $E + F$, $E * F$, and (E). ◀

For example, both 2 and x are expressions by the basis. The inductive step tell us $x + 2$, and $2 * (x + 2)$ are all expressions.

When we have a recursive definition, we can prove theorems about it using the following proof form, which is called structural induction. Let $S(X)$ be a statement about the structures X that are defined by some particular recursive definition.

1. As a basis step, prove $S(X)$ for the basis structure(s) X.
2. For the inductive step, take a structure X that the recursive definition says is formed from $Y_1, Y_2, \cdots, Y_k$. Assume that the statements $S(Y_1), S(Y_2), \cdots, S(Y_k)$ are true, and use these to prove that $S(X)$ is true.

The conclusion is that $S(X)$ is true for all X.

Example Every tree has one more node than it has edges.

Proof The formal statement $S(T)$ is "if T is a tree, and T has n nodes and e edges, then $n = e + 1$."

Basis step: The basis case is when T is a single node. Then $n = 1$ and $e = 0$, so the relationship $n = e + 1$ holds.

Induction step: Let T be a tree built by the inductive step of the definition, from root node N and k smaller trees $T_1, T_2, \cdots, T_k$. Assume that the statements $S(T_i)$ hold for $i = 1, 2, \cdots, k$. i.e., let T_i have n_i nodes and e_i edges; then $n_i = e_i + 1$.

The nodes of T are node N and all the nodes of the T_i's. There are thus $1 + n_1 + n_2 + \cdots + n_k$ nodes in T. The edges of T are the k edges we added explicitly in the inductive definition step, plus the edges of the T_i's.

Hence, T has $k + e_1 + e_2 + \cdots + e_k$ edges. If we substitute $e_i + 1$ for n_i in the count of the number of nodes of T we find that T has $1 + (e_1 + 1) + (e_2 + 1) + \cdots + (e_k + 1) = k + 1 + e_1 + e_2 + \cdots + e_k$ nodes. This expression is exactly 1 more than last one that was given for the number of edges of T. Thus, T has one more node than it has edges. ◀

Question Why structural induction is a valid proof method?

Imagine the recursive definition establishing, on at a time, that contain structure $X_1, X_2, \cdots$ meet the definition. The basis elements come first, and the fact that X_i is in the defined set of structures can only depend on the membership in the defined set of the structures that precede X_i on the list.

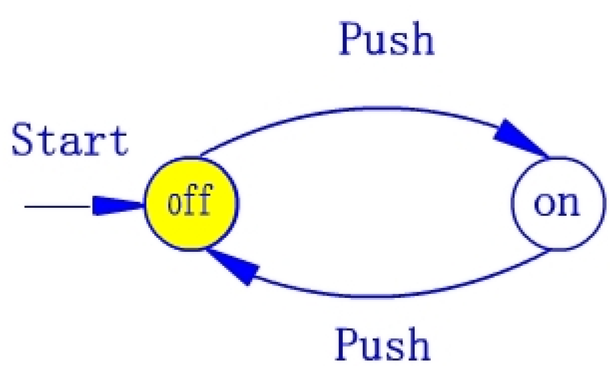
Viewed this way, a structural induction is nothing but an induction on integer n of the statement $S(X_n)$. This induction may be of the generalized form, with multiple basis cases and an inductive step that uses all previous instances of the statement.

Mutual Inductions

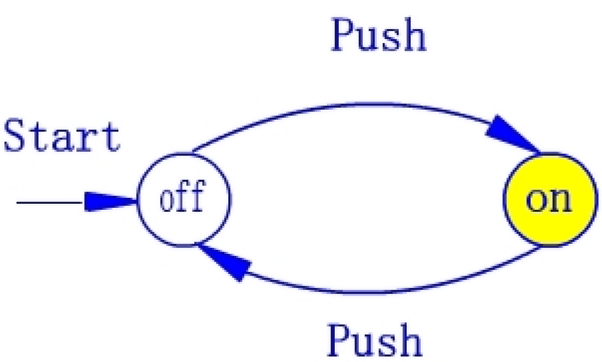
Sometimes, we need to prove a group of statements $S_1(n), S_2(n), \cdots, S_k(n)$ together by induction on n. Automata theory provides many such situations.

In fact, proving a group of statements is no different from proving the conjunction of all the statements. However, when there are really several independent statements to prove, it is generally less confusing to keep the statements separate and to prove them all in their own parts of the basis and inductive steps. We call this sort of proof mutual induction.

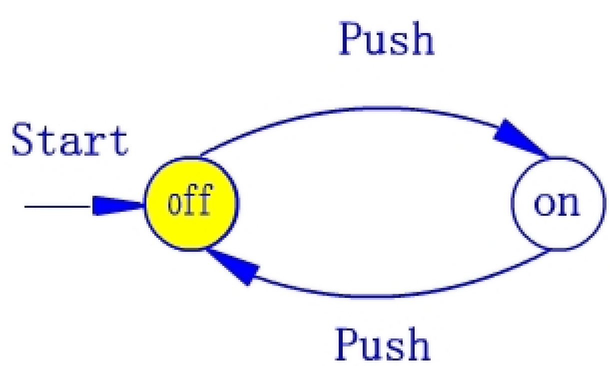
Example Consider an on/off switch, the simplest finite automation. The device remembers whether it is in the *on* state or the *off* state, and it allows the user to press a button whose effect is different, depending on the state of the switch.



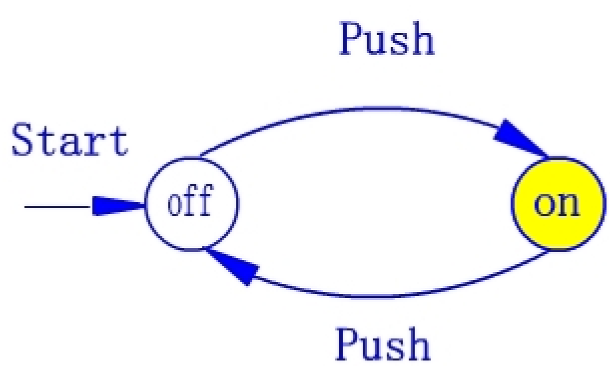
Example Consider an on/off switch, the simplest finite automation. The device remembers whether it is in the *on* state or the *off* state, and it allows the user to press a button whose effect is different, depending on the state of the switch.



Example   Consider an on/off switch, the simplest finite automation. The device remembers whether it is in the *on* state or the *off* state, and it allows the user to press a button whose effect is different, depending on the state of the switch.



Example Consider an on/off switch, the simplest finite automation. The device remembers whether it is in the *on* state or the *off* state, and it allows the user to press a button whose effect is different, depending on the state of the switch.



Since pushing the button switches the state between *on* and *off*, and the switch starts out in the *off* state, we expect that the following statements will together explain the operation of the switch:

$S_1(n)$: The automation is in state *off* after n pushes if and only if n is even.

$S_2(n)$: The automation is in state *on* after n pushes if and only if n is odd.

Proof We give the basis and inductive parts of the proofs of the statements $S_1(n)$ and $S_2(n)$ below.

Since there are two statements, each of which must be proved in both directions, there are actually four cases to the basis, and four cases to the induction as well.

Basis step: For the basis, we choose $n = 0$.

1. [$S_1(0)$; If] Since 0 is in fact even, we must show that after 0 pushes, the automation is in state *off*. Since that is the start state, the automation is indeed in state *off* after 0 pushes.

2. [$S_1(0)$; Only-if] The automation is in state *off* after 0 pushes, so we must show that 0 is even. But 0 is even by definition, so there is nothing more to prove.

3. [$S_2(0)$; If] The hypothesis of the "if" part of S_2 is that 0 is odd. Since the hypothesis is false, the if-then statement is true. Thus, this part of the basis also holds.

4. [$S_2(0)$; Only-if] The hypothesis is also false, since the only way to get to state *on* is by following an arc labeled *Push*, which requires that the button be pushed at least once. Since the hypothesis is false, we can again conclude that the if-then statement is true.

Induction step: Now, we assume that $S_1(n)$ and $S_2(n)$ are true, and try to prove that $S_1(n+1)$ and $S_2(n+1)$ are true.

1. [$S_1(n+1)$; If] The hypothesis for this part is that $n+1$ is even. Thus, n is odd. The "if" part of statement $S_2(n)$ says that after n pushes, the automation is in state *on*. The arc from *on* to *off* labeled *Push* tells us that the $(n+1)$st push will cause the automation to enter state *off*. That completes the proof of the "if" part of $S_1(n+1)$.

2. [$S_1(n+1)$; Only-if] The hypothesis is that the automation is in state *off* after $n+1$ pushes. Inspecting the automation tells us that the only way to get to state *off* after one or more moves is to be in state *on* and receive an input *Push*. Thus, if we are in state *off* after $n+1$ pushes, we must have been in state *on* after n pushes.

Then, we may use the “only-if” part of statement $S_2(n)$ to conclude that n is odd. Consequently, $n + 1$ is even, which is the desired conclusion for the only-if portion of $S_1(n + 1)$.

3. [$S_2(n + 1)$; If] This part is essentially the same as part (1), with the roles of statements S_1 and S_2 exchanged, and with the roles of “odd” and “even” exchanged.

4. [$S_2(n + 1)$; Only-if] This part is essentially the same as part (2), with the roles of statements S_1 and S_2 exchanged, and with the roles of “odd” and “even” exchanged.

◀

Break for 15 minutes

The Central Concepts of Automata Theory

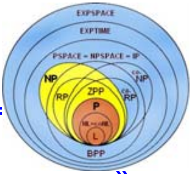
Languages

We start with a finite, nonempty set Σ of symbols, called the alphabet.

Example $\Sigma = \{0, 1\}$, the binary alphabet.

Example $\Sigma = \{a, b, \cdots, z\}$, the set of all lower-case letters.

Example The set of all ASCII characters, or the set of all printable ASCII characters.



From the individual symbols we construct strings, which are finite sequence of symbols from the alphabet.

Example 00011101011 is a string from the binary alphabet $\Sigma = \{0, 1\}$.

We will need to refer to the empty string, which is a string with zero occurrences of symbols from an alphabet.

✍ The empty sting is denoted ϵ.

The concatenation of strings x and y, denoted xy, is the string obtained by appending the symbols of y to the right end of x, that is, if $x = a_1a_2\cdots a_i$, $y = b_1b_2\cdots b_j$, then $xy = a_1a_2\cdots a_ib_1b_2\cdots b_j$.

Example Let $x = 01101$, and $y = 110$. Then $xy = 01101110$.

Notice that: For any string w, $\epsilon w = x\epsilon = w$.

The length of a strings is the number of positions for symbols in the string.

✍ The length of a sting w is denoted $|w|$. For example, $|00110| = 5$, $|\epsilon| = 0$.

The powers of an alphabet is a set of strings of a certain length from an alphabet.

✍ The set of strings of length k, each of whose symbols is in Σ, is denoted Σ^k. For example, if $\Sigma = \{0, 1\}$, then $\Sigma^0 = \{\epsilon\}$, $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$.

✍ The set of all strings over an alphabet Σ is denoted Σ^*. For instance, $\{0, 1\}^* = \{\epsilon, 0, 1, 00, 10, 01, 11, 000, \cdots\}$.

✍ The set of nonempty strings from alphabet Σ is denoted Σ^+.

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$

- $\Sigma^* = \Sigma^+ \cup \{\epsilon\} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$

- The set Σ^* of strings over the alphabet Σ can be defined recursively:

  Basis step: $\epsilon \in \Sigma^*$.

  Inductive step: if $w \in \Sigma^*$ and $x \in \Sigma$, then $wx \in \Sigma^*$.

- The length of a string $|w|$ can be also recursively defined by

$$|\epsilon| = 0; \qquad |wx| = |w| + 1 \text{ if } w \in \Sigma^* \text{ and } x \in \Sigma.$$

Question   Prove $|xy| = |x| + |y|$, where x and y belong to Σ^*.

A language is defined as a subset of Σ^*, where Σ is a particular alphabet. A string in a language L will be called a sentence of L.

Example   The set of legal English words is a language over the alphabet that consists of all the letters.

Example   The set of legal C programs is a language over the set of all ASCII characters.

Example The language of all strings consisting of n 0's followed by n 1's, for some $n \geq 0$:

$$\{\epsilon, 01, 0011, 000111, \cdots\}.$$

Example The language of strings of 0's and 1's with an equal number of each:

$$\{\epsilon, 01, 10, 0011, 0101, 1001, \cdots\}.$$

Example The language of binary numbers whose value is a prime:

$$\{10, 11, 101, 111, 1011, \cdots\}.$$

- For any alphabet Σ, Σ^* is a language.
- $\emptyset$, the empty language, is a language over any alphabet.
- $\{\epsilon\}$, the language consisting of only the empty string, is also a language over any alphabet.

Notice that: $\emptyset \neq \{\epsilon\}$.

- It is common to describe a language using a “set-former”:

$$\{w| \text{ somthing about } w\}.$$

  For example, $\{w| \ w \text{ consists of an equal number of } 0'\text{s and } 1'\text{s}\}$.

- It is also common to replace w by some expression with parameters and describe the strings in the language by stating conditions on the parameters. For example, $\{0^i1^j | 0 \leq i \leq j\}$, and $\{0^n1^n | n \geq 1\}$ etc.

Since languages are sets, the union, intersection, and difference of two languages are immediately defined. Let L and M are both languages, there are other two operations on languages.

- Concatenation (dot) $\quad L.M = \{w \mid w = xy, x \in L, y \in M\}$
- Closure (star) $\quad L^* = \bigcup_{i=0}^{\infty} L^i,$

  where $L^0 = \{\epsilon\},\ \ L^1 = L,\ \ L^{k+1} = L.L^k,\ \ (k = 1, 2, \cdots)$

☞ The idea of the closure of a language is somewhat tricky!

Question

- $L = \{0, 11\}, \quad L^* = ?$
- $L = \{\text{all strings of } 0's\}, \quad L^* = ?$
- $L = \emptyset, \; L^* = ?$
- $L = \{\epsilon\}, \; L^* = ?$

☞ The idea of the closure of a language is somewhat tricky!

Question

- $L = \{0, 11\}, \quad L^* = \{\epsilon, 0, 11, 00, 011, 110, 1111, 000, 0011, 0110, 01111, \cdots\}$
- $L = \{\text{all strings of } 0's\}, \quad L^* = ?$
- $L = \emptyset, \ L^* = ?$
- $L = \{\epsilon\}, \ L^* = ?$

☞ The idea of the closure of a language is somewhat tricky!

Question

- $L = \{0, 11\}, \quad L^* = \{\epsilon, 0, 11, 00, 011, 110, 1111, 000, 0011, 0110, 01111, \cdots\}$
- $L = \{\text{all strings of } 0's\}, \quad L^* = \{\text{all strings of } 0's\}$
- $L = \emptyset, \ L^* = ?$
- $L = \{\epsilon\}, \ L^* = ?$

☞ The idea of the closure of a language is somewhat tricky!

Question

- $L = \{0, 11\}, \quad L^* = \{\epsilon, 0, 11, 00, 011, 110, 1111, 000, 0011, 0110, 01111, \cdots\}$
- $L = \{\text{all strings of } 0's\}, \quad L^* = \{\text{all strings of } 0's\}$
- $L = \emptyset, \;\; L^* = \{\epsilon\}$
- $L = \{\epsilon\}, \;\; L^* = ?$

☞ The idea of the closure of a language is somewhat tricky!

Question

- $L = \{0, 11\}, \quad L^* = \{\epsilon, 0, 11, 00, 011, 110, 1111, 000, 0011, 0110, 01111, \cdots\}$
- $L = \{\text{all strings of } 0's\}, \quad L^* = \{\text{all strings of } 0's\}$
- $L = \emptyset, \ L^* = \{\epsilon\}$
- $L = \{\epsilon\}, \ L^* = \{\epsilon\}$

Let L, M, N are all languages, we have the following algebraic laws.

- $L.(M.N) = (L.M).N$

  Concatenation is *associative*.

- $\{\epsilon\}.L = L.\{\epsilon\} = L$

  $\{\epsilon\}$ is *identity* for concatenation.

- $\emptyset.L = L.\emptyset = \emptyset$

  $\emptyset$ is *annihilator* for concatenation.

- $L.(M \cup N) = L.M \cup L.N$

  Concatenation is *left distributive* over union.

- $(M \cup N).L = M.L \cup N.L$

  Concatenation is *right distributive* over union.

- $(L^*)^* = L^*$

  Closure is *idempotent*.

Question Is concatenation communicative, i.e. $L.M = M.L$?

Grammars

A grammar is a quadruple $G = (V, T, P, S)$.

- V is a finite set of *variables*.
- T is a finite set of *terminals symbols*.
- P is a finite set of *productions* of the form $x \to y$, where $x \in (V \cup T)^+$ and $y \in (V \cup T)^*$.
- S $(\in V)$ is a designated variable called the *start symbol*.

Now we develop the notation for describing the derivations.

Let $G = (V, T, P, S)$ be a grammar, $\{\alpha, \beta\} \subset (V \cup T)^*$, and $x \to y \in P$. Then we write

$$\alpha x \beta \underset{G}{\Rightarrow} \alpha y \beta$$

or, if G is understood

$$\alpha x \beta \Rightarrow \alpha y \beta$$

and say that $\alpha x \beta$ derives $\alpha y \beta$.

Specially, we have $x \Rightarrow y$.

We may extend the $\Rightarrow$ relationship to present zero, one, or many derivation steps.

In other words, we define $\overset{*}{\Rightarrow}$ to be the reflexive and transitive closure of $\Rightarrow$, as follows:

Basis step: Let $\alpha \in (V \cup T)^*$. Then $\alpha \overset{*}{\Rightarrow} \alpha$.

Inductive step: If $\alpha \overset{*}{\Rightarrow} \beta$, and $\beta \Rightarrow \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$.

☞ Use induction we can prove that if $\alpha \overset{*}{\Rightarrow} \beta$, and $\beta \overset{*}{\Rightarrow} \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$.

Let $G(V, T, P, S)$ be a grammar. Then the set

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

is the language generated by G.

If $w \in L(G)$, then the sequence $S \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_n \Rightarrow w$ is a derivation of the sentence w. The strings $S, w_1, \cdots, w_n$, which contain variables as well as terminals, are called sentential forms of the derivation.

In general, we call $L = \{w \in T^* \mid A \overset{*}{\underset{G}{\Rightarrow}} w\}$ the language of variable A if $A \in V$.

Example Consider the grammar $G = (\{S\}, \{a, b\}, P, S)$, with P given by

$$S \to aSb, \quad S \to \epsilon.$$

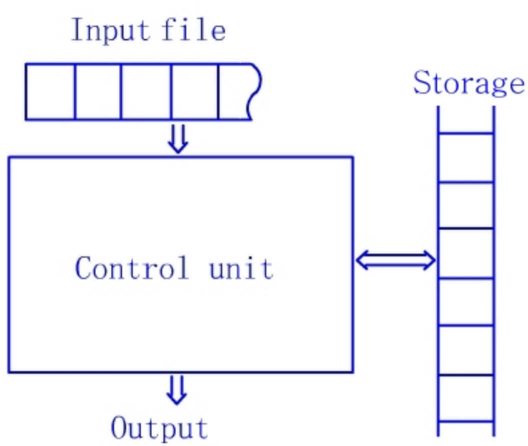
The string $aabb$ is a sentence in the language generated by G.

A grammar G completely defines $L(G)$, but it may not be easy to get a very explicit description of the language from the grammar.

Here, however, the answer is fairly clear. It is not hard to conjecture that $L(G) = \{a^n b^n \mid n \geq 0\}$, and it is easy to prove it.

Automata

An automaton is an abstract model of a digital computer. Following figure shows a schematic representation of a general automaton.



This general model covers all the automata we will discuss in this course. A finite-state control will be common to all specific cases, but differences will arise from the way in which the output can be produced and the nature of the temporary storage.

It is necessary to distinguish between deterministic automata and nondeterministic automata. A deterministic automaton is one in which each move in uniquely determined by current configuration. In a nondeterministic automaton, this is not so.

Thank you!