Introduction to the Theory of Computation

Jingde Cheng

Saitama University

Introduction to the Theory of Computation

- Enumerability and Diagonalization
- Finite Automata and Regular Languages
- Pushdown Automata and Context-Free Languages
- Computation: Turing Machines
- Computation: Turing-Computability (Turing-Decidability)
- Computation: Reducibility (Turing-Reducibility)
- Computation: Recursive Functions
- Computation: Recursive Sets and Relations
- Equivalent Definitions of Computability
- Advanced Topics in Computability Theory
- Computational Complexity
- Time Complexity
- Space Complexity
- Intractability
- Advanced Topics in Complexity Theory

Automata Theory: What Is It and Why Study It? [HMU-ToC-07]

♣ Automata Theory: What is it?

- Automata theory is the study of abstract computing devices, or "machines".
- In the 1940's and 1950's, a number of researchers studied some kinds of "machines", called "*finite automata*" today, that are simpler than Turing's Turing machines.

♣ Automata Theory: Why study it?

- Finite automata are a useful model for many important kinds of hardware and software.
- There are many systems that may be viewed as being at all times in one of a finite number of "states"; these systems can be represented and analyzed by finite automata.

(Deterministic) Finite Automata (DFAs): An Example [S-ToC-13]

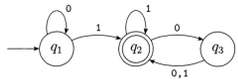


FIGURE **1.4**
A finite automaton called M_1 that has three states

Figure 1.4 is called the ***state diagram*** of M_1. It has three ***states***, labeled q_1, q_2, and q_3. The ***start state***, q_1, is indicated by the arrow pointing at it from nowhere. The ***accept state***, q_2, is the one with a double circle. The arrows going from one state to another are called ***transitions***.

For example, when we feed the input string 1101 to the machine M_1 in Figure 1.4, the processing proceeds as follows.

1. Start in state q_1.
2. Read 1, follow transition from q_1 to q_2.
3. Read 1, follow transition from q_2 to q_2.
4. Read 0, follow transition from q_2 to q_3.
5. Read 1, follow transition from q_3 to q_2.
6. *Accept* because M_1 is in an accept state q_2 at the end of the input.

(Deterministic) Finite Automata (DFAs) as Machines [LP-ToC-98]

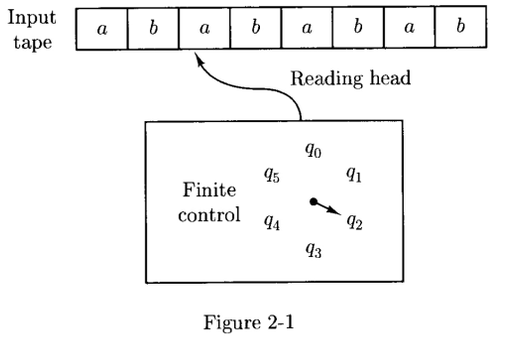


Figure 2-1

(Deterministic) Finite Automata (DFAs) as Machines [LP-ToC-98]

Let us now describe the operation of a finite automaton in more detail. Strings are fed into the device by means of an **input tape**, which is divided into squares, with one symbol inscribed in each tape square (see Figure 2-1). The main part of the machine itself is a "black box" with innards that can be, at any specified moment, in one of a finite number of distinct internal **states**. This black box —called the **finite control**— can sense what symbol is written at any position on the input tape by means of a movable **reading head**. Initially, the reading head is placed at the leftmost square of the tape and the finite control is set in a designated **initial state**. At regular intervals the automaton reads one symbol from the input tape and then enters a new state *that depends only on the current state and the symbol just read* —this is why we shall call this device a *deterministic* finite automaton, to be contrasted to the *nondeterministic* version introduced in the next section. After reading an input symbol, the reading head moves one square to the right on the input tape so that on the next move it will read the symbol in the next tape square. This process is repeated again and again; a symbol is read, the reading head moves to the right, and the state of the finite control changes. Eventually the reading head reaches the end of the input string. The automaton then indicates its approval or disapproval of what it has read by the state it is in at the end: if it winds up in one of a set of **final states** the input string is considered to be **accepted**. The **language accepted** by the machine is the set of strings it accepts.

(Deterministic) Finite Automata (DFAs): Formal Definition

DEFINITION 1.5

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,[1]
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.[2]

- 1 *Transition function* δ is a function from the Cartesian (direct) product ($Q \times \Sigma$) of Q and Σ to Q.
- 2 *Accept states* are also called *final states*.

DFAs: Transition Diagrams and Tables [HMU-ToC-07]

Transition Diagrams

A *transition diagram* for a DFA $A = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

a) For each state in Q there is a node.

b) For each state q in Q and each input symbol a in Σ, let $\delta(q, a) = p$. Then the transition diagram has an arc from node q to node p, labeled a. If there are several input symbols that cause transitions from q to p, then the transition diagram can have one arc, labeled by the list of these symbols.

c) There is an arrow into the start state q_0, labeled *Start*. This arrow does not originate at any node.

d) Nodes corresponding to accepting states (those in F) are marked by a double circle. States not in F have a single circle.

Transition Tables

A *transition table* is a conventional, tabular representation of a function like δ that takes two arguments and returns a value. The rows of the table correspond to the states, and the columns correspond to the inputs. The entry for the row corresponding to state q and the column corresponding to input a is the state $\delta(q, a)$.

DFAs: An Example [S-ToC-13]

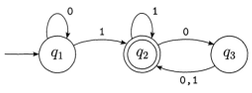


FIGURE 1.6
The finite automaton M_1

We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state, and
5. $F = \{q_2\}$.

Alphabets, Strings, and Languages

♣ **Alphabet**

- [D] An *alphabet* is a non-empty finite set of symbols.
- We generally use capital Greek letters Σ and Γ to designate alphabets.

♣ **Examples of alphabet**

- Σ_1 = {0, 1}
- Σ_2 = {a, b, c, …, z}
- Γ = {0, 1, x, y, z}

Alphabets, Strings, and Languages

♣ **String**

- [D] A *string* over an alphabet is a finite sequence of symbols from that alphabet, usually written next to one another and not separated by commas.
- For a given alphabet Σ, Σ^* denotes all strings over Σ (The reflexive transitive closure of symbol connection relation).

♣ **The length of a string**

- [D] If w is string, the length of w, written $|w|$, is the number of its symbols.
- Note: A symbol may appear in a strings many times.

♣ **The empty string**

- [D] The string of length zero is called *the empty string* and is written ε.

Alphabets, Strings, and Languages

♣ **The reverse of string**

- [D] For string $w = w_1w_2w_3\ldots w_n$, *the reverse* of w, written w^R, is the string obtained by writing w in the opposite order, i.e., $w^R = w_n\ldots w_3w_2w_1$.

♣ **The concatenation of strings**

- [D] For string x of length m and string y of length n, *the concatenation* of x and y, written xy, is the string obtained by appending y to the end of x, as in $x_1x_2x_3\ldots x_my_1y_2y_3\ldots y_n$.
- [D] To concatenate a string with itself many time, say k times, we use the superscript notation x^k.

Alphabets, Strings, and Languages

♣ **Prefix of string**

- **[D] For string y, we say that string x is *a prefix* of y if a string z exists where $xz = y$, and that x is *a proper prefix* of y if in addition $x \neq y$.**
- **Note: Any string is a prefix of itself.**

♣ **Language**

- **[D] For a given alphabet Σ, *a language over Σ*, denoted by L_Σ, is a set of strings over Σ.**
- **For any Σ, $L_\Sigma \subseteq \Sigma^*$, $L_\Sigma \cup L_\Sigma{}^C = \Sigma^*$, $L_\Sigma \cap L_\Sigma{}^C = \varnothing$.**
- **[D] A language is *prefix-free* if no member is a proper prefix of another member.**

DFAs: Regular Languages

♣ **Languages of finite automata**

- **[D] If A is the set of all strings that finite automaton M accepts, we say that A is *the language of M* and write *$L(M)$* = A, also say that M *recognizes* A or that M *accepts* A.**

♣ **Notes**

- **A finite automaton may accept several strings, but it always recognizes only one language.**
- **If the finite automaton accepts no strings, it still recognizes one language, namely, *the empty language* ∅.**

♣ **Regular language**

- **[D] A language is called a *regular language* if some finite automaton recognizes it.**

Deterministic Finite Accepter (DFAs): Formal Definition [L-ToC-17]

DEFINITION 2.1

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

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

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

DEFINITION 2.2

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

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

DEFINITION 2.3

A language L is called **regular** if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

DFA Examples [S-ToC-13]

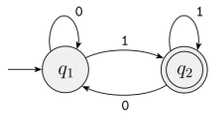


FIGURE 1.8
State diagram of the two-state finite automaton M_2

In the formal description, M_2 is $(\{q_1, q_2\}, \{0,1\}, \delta, q_1, \{q_2\})$. The transition function δ is

	0	1
q_1	q_1	q_2
q_2	q_1	q_2.

DFA Examples [S-ToC-13]

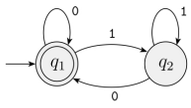


FIGURE 1.10
State diagram of the two-state finite automaton M_3

Machine M_3 is similar to M_2 except for the location of the accept state. As usual, the machine accepts all strings that leave it in an accept state when it has finished reading. Note that because the start state is also an accept state, M_3 accepts the empty string ε. As soon as a machine begins reading the empty string, it is at the end; so if the start state is an accept state, ε is accepted. In addition to the empty string, this machine accepts any string ending with a 0. Here,

$$L(M_3) = \{w | \ w \text{ is the empty string } \varepsilon \text{ or ends in a } 0\}.$$

DFA Examples [S-ToC-13]

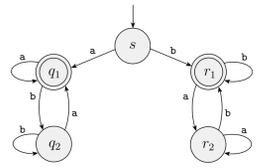


FIGURE 1.12
Finite automaton M_4

Machine M_4 has two accept states, q_1 and r_1, and operates over the alphabet $\Sigma = \{a, b\}$. Some experimentation shows that it accepts strings a, b, aa, bb, and bab, but not strings ab, ba, or bbba. This machine begins in state s, and after it reads the first symbol in the input, it goes either left into the q states or right into the r states. In both cases, it can never return to the start state (in contrast to the previous examples), as it has no way to get from any other state back to s. If the first symbol in the input string is a, then it goes left and accepts when the string ends with an a. Similarly, if the first symbol is a b, the machine goes right and accepts when the string ends in b. So M_4 accepts all strings that start and end with a or that start and end with b. In other words, M_4 accepts strings that start and end with the same symbol.

DFA Examples [S-ToC-13]

Figure 1.14 shows the three-state machine M_5, which has a four-symbol input alphabet, $\Sigma = \{\langle \text{RESET} \rangle, 0, 1, 2\}$. We treat $\langle \text{RESET} \rangle$ as a single symbol.

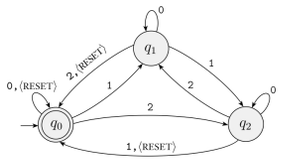


FIGURE 1.14
Finite automaton M_5

Machine M_5 keeps a running count of the sum of the numerical input symbols it reads, modulo 3. Every time it receives the ⟨RESET⟩ symbol, it resets the count to 0. It accepts if the sum is 0 modulo 3, or in other words, if the sum is a multiple of 3. ■

DFA Examples [L-ToC-17]

The graph in Figure 2.1 represents the dfa

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

where δ is given by

$$\begin{aligned} \delta(q_0, 0) &= q_0, & \delta(q_0, 1) &= q_1, \\ \delta(q_1, 0) &= q_0, & \delta(q_1, 1) &= q_2, \\ \delta(q_2, 0) &= q_2, & \delta(q_2, 1) &= q_1. \end{aligned}$$

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

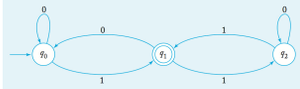


FIGURE 2.1

DFA Examples [L-ToC-17]

Consider the dfa in Figure 2.2.

In drawing Figure 2.2 we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in Figure 2.2 remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the dfa goes into state q_2, from which it can never escape. The state q_2 is a **trap state**. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

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

FIGURE 2.2

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

FIGURE 2.3

DFA Examples [L-ToC-17]

Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab.

The only issue here is the first two symbols in the string; after they have been read, no further decisions are needed. Still, the automaton has to process the whole string before its decision is made. We can therefore solve the problem with an automaton that has four states: an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is an a and the second is a b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not an a or the second one is not a b, the automaton enters the nonfinal trap state. The simple solution is shown in Figure 2.4.

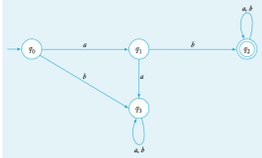


FIGURE 2.4

Find a dfa that accepts all the strings on $\{0,1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0's. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0's were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of the string to the left, for example, whether or not the two previous symbols were 00. If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00$$

because this situation arises only if there are three consecutive 0's. We are only interested in the last two, a fact we remember by keeping the dfa in the state 00. A complete solution is shown in Figure 2.5. We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct.

FIGURE 2.5

DFA Examples [L-ToC-17]

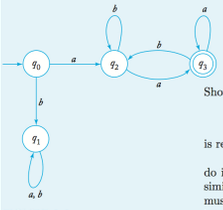


FIGURE 2.6

Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a dfa for it. The construction of a dfa for this language is similar to Example 2.3, but a little more complicated. What this dfa must do is check whether a string begins and ends with an a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string. This difficulty is overcome by simply putting the dfa into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the dfa out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in Figure 2.6. Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the dfa accepts a string if and only if it begins and ends with an a. Since we have constructed a dfa for the language, we can claim that, by definition, the language is regular.

DFA Examples [L-ToC-17]

Let L be the language in Example 2.5. Show that L^2 is regular. Again we show that the language is regular by constructing a dfa for it. We can write an explicit expression for L^2, namely,

$$L^2 = \{aw_1aaw_2a : w_1, w_2 \in \{a, b\}^*\}.$$

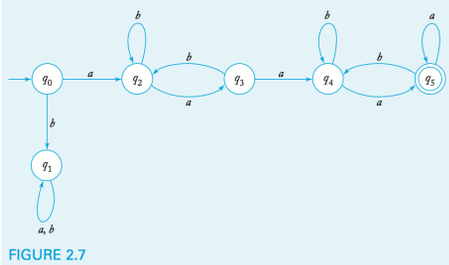


FIGURE 2.7

DFAs: Formal Definition of Computation

♣ **Formal definition of computation by a finite automata**

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M ***accepts*** w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$, and
3. $r_n \in F$.

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state. We say that M ***recognizes language*** A if $A = \{w \mid M \text{ accepts } w\}$.

♣ **Regular language**

- **[D] A language is called a *regular language* if some finite automaton recognizes it.**

DFA Examples [S-ToC-13]

Figure 1.14 shows the three-state machine M_5, which has a four-symbol input alphabet, $\Sigma = \{\langle\text{RESET}\rangle, 0, 1, 2\}$. We treat $\langle\text{RESET}\rangle$ as a single symbol.

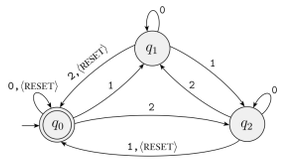


FIGURE 1.14
Finite automaton M_5

Machine M_5 keeps a running count of the sum of the numerical input symbols it reads, modulo 3. Every time it receives the $\langle\text{RESET}\rangle$ symbol, it resets the count to 0. It accepts if the sum is 0 modulo 3, or in other words, if the sum is a multiple of 3.

DFAs: Formal Definition of Computation

♣ **Example: Machine M_5**

- **Let *w* be the string 10⟨RESET⟩22⟨RESET⟩012.**
- **Then M_5 accepts *w* according to the formal definition of computation because the sequence of states it enters when computing on *w* is**
  $q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0$,
  which satisfies the three conditions.
- **The language of M_5 is**
  $L(M_5)$ = {*w* | the sum of the symbols in *w* is 0 modulo 3, except that ⟨RESET⟩ resets the count to 0}.
- **As M_5 recognizes this language, it is a regular language.**

Designing DFAs: An Example [S-ToC-13]

- **Suppose that the alphabet is {0, 1} and that the language consists of all strings with an odd number of 1s.**
- **Let us construct a finite automaton E_1 to recognize this language.**
- **We start getting an input string of 0s and 1s symbol by symbol.**
- **We do not need to remember the entire string seen so far in order to determine whether the number of 1s is odd.**
- **We just need to remember whether the number (answer) of 1s seen so far is even or odd and keep track of this information as we read new symbols.**
- **If we read a 1, flip the answer (even to odd, or odd to even); but if we read a 0, leave the answer as is.**

Designing DFAs: An Example [S-ToC-13]

- **Once we have determined the necessary information to remember about the string as it is being read, we represent this information as a finite list of possibilities.**
- **In this instance, the possibilities would be:**
  1. even so far, and
  2. odd so far.

FIGURE 1.18
The two states q_{even} and q_{odd}

- **Next, we assign the transitions by seeing how to go from one possibility to another upon reading a symbol.**
- **So, if state q_{even} represents the even possibility and state q_{odd} represents the odd possibility, we would set the transitions to flip state on a 1 and stay put on a 0.**

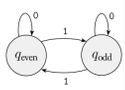
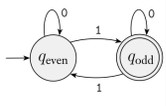


FIGURE 1.19
Transitions telling how the possibilities rearrange

Designing DFAs: An Example [S-ToC-13]

- **Next, we set the start state to be the state corresponding to the possibility associated with having seen 0 symbols so far (the empty string ε).**
- **In this case, the start state corresponds to state q_{even} because 0 is an even number.**
- **Last, set the accept states to be those corresponding to possibilities where we want to accept the input string.**
- **Set q_{odd} to be an accept state because we want to accept when we have seen an odd number of 1s.**



- **These additions are shown in the figure.**

FIGURE 1.20
Adding the start and accept states

Designing DFAs: An Example [S-ToC-13]

EXAMPLE 1.21

This example shows how to design a finite automaton E_2 to recognize the regular language of all strings that contain the string 001 as a substring. For example, 0010, 1001, 001, and 11111110011111 are all in the language, but 11 and 0000 are not. How would you recognize this language if you were pretending to be E_2? As symbols come in, you would initially skip over all 1s. If you come to a 0, then you note that you may have just seen the first of the three symbols in the pattern 001 you are seeking. If at this point you see a 1, there were too few 0s, so you go back to skipping over 1s. But if you see a 0 at that point, you should remember that you have just seen two symbols of the pattern. Now you simply need to continue scanning until you see a 1. If you find it, remember that you succeeded in finding the pattern and continue reading the input string until you get to the end.

So there are four possibilities: You

1. haven't just seen any symbols of the pattern,
2. have just seen a 0,
3. have just seen 00, or
4. have seen the entire pattern 001.

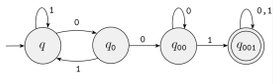


FIGURE 1.22
Accepts strings containing 001

Assign the states q, q_0, q_{00}, and q_{001} to these possibilities. You can assign the transitions by observing that from q reading a 1 you stay in q, but reading a 0 you move to q_0. In q_0 reading a 1 you return to q, but reading a 0 you move to q_{00}. In q_{00} reading a 1 you move to q_{001}, but reading a 0 leaves you in q_{00}. Finally, in q_{001} reading a 0 or a 1 leaves you in q_{001}. The start state is q, and the only accept state is q_{001}, as shown in Figure 1.22.

The Regular Operations: Formal Definition

♣ **Regular Operations**

DEFINITION 1.23

Let A and B be languages. We define the regular operations ***union***, ***concatenation***, and ***star*** as follows:

- **Union**: $A \cup B = \{x | \ x \in A \text{ or } x \in B\}$.
- **Concatenation**: $A \circ B = \{xy | \ x \in A \text{ and } y \in B\}$.
- **Star**: $A^* = \{x_1 x_2 \ldots x_k | \ k \geq 0 \text{ and each } x_i \in A\}$.

♣ **Notes**

- **The concatenation operation attaches a string from A in front of a string from B in all possible ways to get the strings in the new language.**
- **The star operation works by attaching any number of strings in A together to get a string in the new language. The *empty string* ε is always a member of A^*, no matter what A is.**

The Regular Operations: Theorems

♣ **Closed under operation**

- **[D] Generally speaking, a collection of objects is *closed* under some operation if applying that operation to members of the collection returns an object still in the collection.**

♣ **Three theorems**

THEOREM 1.25

The class of regular languages is closed under the union operation.

In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

THEOREM 1.26

The class of regular languages is closed under the concatenation operation.

In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

THEOREM 1.49

The class of regular languages is closed under the star operation.

Non-deterministic Finite Automata (NFAs): An Example [S-ToC-13]

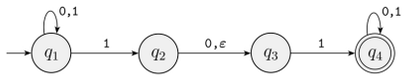


FIGURE 1.27
The nondeterministic finite automaton N_1

The difference between a deterministic finite automaton, abbreviated DFA, and a nondeterministic finite automaton, abbreviated NFA, is immediately apparent. First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. The NFA shown in Figure 1.27 violates that rule. State q_1 has one exiting arrow for 0, but it has two for 1; q_2 has one arrow for 0, but it has none for 1. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol.

Second, in a DFA, labels on the transition arrows are symbols from the alphabet. This NFA has an arrow with the label ε. In general, an NFA may have arrows labeled with members of the alphabet or ε. Zero, one, or many arrows may exit from each state with the label ε.

Non-deterministic Finite Automata (NFAs): An Example [S-ToC-13]

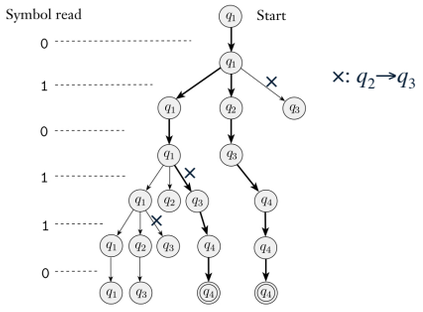


FIGURE 1.29
The computation of N_1 on input 010110

Non-deterministic Finite Automata (NFAs): Formal Definition

DEFINITION 1.37

A ***nondeterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta: Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

- **For any alphabet Σ we write Σ_ε to be $\Sigma \cup \{\varepsilon\}$.**
- **$P(Q)$ (2^Q) is the power set of Q.**
- **Important fact: Every NFA can be converted into an equivalent DFA.**

Non-deterministic Finite Accepters (NFAs): Formal Definition [L-ToC-17]

DEFINITION 2.4

A **nondeterministic finite accepter** or **nfa** is defined by the quintuple

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

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

- **λ is the empty string, 2^Q is the power set of Q.**

DEFINITION 2.5

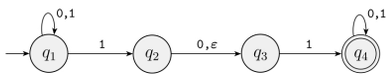
For an nfa, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

NFA vs. DFA: Differences between Their Definitions

- **Multiple next states**
  - **In an NFA, the range of transition function δ is the power set $P(Q)$ (2^Q), so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by a transition.**
- **Allowing the empty string as the second argument of δ**
  - **The empty string ε is allowed as the second argument of transition function δ. This means that the NFA can make a situation without consuming an input symbol.**
- **No next state**
  - **In an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.**

NFAs: An Example [S-ToC-13]

Recall the NFA N_1:



The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0,1\}$,
3. δ is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$,

4. q_1 is the start state, and
5. $F = \{q_4\}$.

NFA Examples [S-ToC-13]

Let A be the language consisting of all strings over {0,1} containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not). The following four-state NFA N_2 recognizes A.

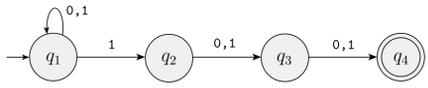


FIGURE 1.31
The NFA N_2 recognizing A

One good way to view the computation of this NFA is to say that it stays in the start state q_1 until it "guesses" that it is three places from the end. At that point, if the input symbol is a 1, it branches to state q_2 and uses q_3 and q_4 to "check" on whether its guess was correct.

NFA Examples [S-ToC-13]

As mentioned, every NFA can be converted into an equivalent DFA; but sometimes that DFA may have many more states. The smallest DFA for A contains eight states. Furthermore, understanding the functioning of the NFA is much easier, as you may see by examining the following figure for the DFA.

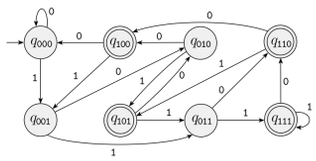


FIGURE 1.32
A DFA recognizing A

Suppose that we added ε to the labels on the arrows going from q_2 to q_3 and from q_3 to q_4 in machine N_2 in Figure 1.31. So both arrows would then have the label 0, 1, ε instead of just 0, 1. What language would N_2 recognize with this modification? Try modifying the DFA in Figure 1.32 to recognize that language.

NFA Examples [S-ToC-13]

The following NFA N_3 has an input alphabet {0} consisting of a single symbol. An alphabet containing only one symbol is called a ***unary alphabet***.

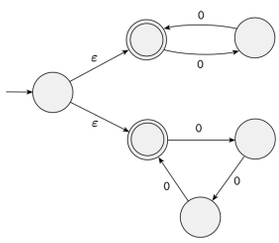


FIGURE 1.34
The NFA N_3

NFA Examples [S-ToC-13]

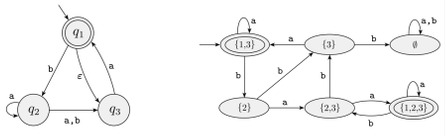


FIGURE 1.36
The NFA N_4

FIGURE 1.44
DFA D after removing unnecessary states

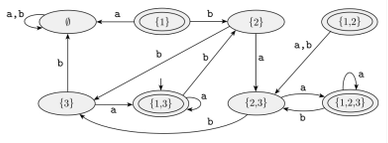


FIGURE 1.43
A DFA D that is equivalent to the NFA N_4

NFA Examples [L-ToC-17]

Consider the transition graph in Figure 2.8. It describes a nondeterministic accepter since there are two transitions labeled a out of q_0.

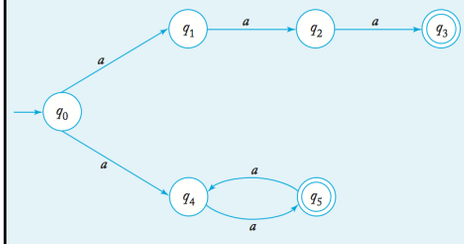


FIGURE 2.8

NFA Examples [L-ToC-17]

A nondeterministic automaton is shown in Figure 2.9. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.

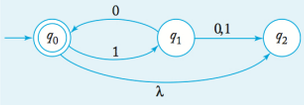


FIGURE 2.9

NFA Examples [L-ToC-17]

Figure 2.10 represents an nfa. It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

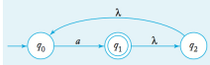


FIGURE 2.10

Non-deterministic Finite Accepters (NFAs): Formal Definition [L-ToC-17]

DEFINITION 2.6

The language L accepted by an nfa $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

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

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

DEFINITION 2.7

Two finite accepters, M_1 and M_2, are said to be equivalent if

$$L(M_1) = L(M_2),$$

that is, if they both accept the same language.

THEOREM 2.2

Let L be the language accepted by a nondeterministic finite accepter $M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Then there exists a deterministic finite accepter $M_D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that

$$L = L(M_D).$$

Equivalence of NFAs and DFAs

- ♣ **Equivalence**
  - ◆ **[D] We say that two FAs are *equivalent* if they recognize the same language.**
  - ◆ **Important fact: DFAs and NFAs recognize the same class of languages.**
- ♣ **Equivalence theorem**

THEOREM 1.39

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

COROLLARY 1.40

A language is regular if and only if some nondeterministic finite automaton recognizes it.

Equivalence of NFAs and DFAs [L-ToC-17]

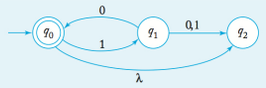


FIGURE 2.9

The dfa shown in Figure 2.11 is equivalent to the nfa in Figure 2.9 since they both accept the language $\{(10)^n : n \geq 0\}$.

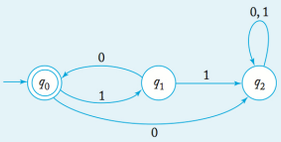


FIGURE 2.11

Equivalence of NFAs and DFAs [L-ToC-17]

Convert the nfa in Figure 2.12 to an equivalent dfa. The nfa starts in state q_0, so the initial state of the dfa will be labeled $\{q_0\}$. After reading an a, the nfa can be in state q_1 or, by making a λ-transition, in state q_2. Therefore, the corresponding dfa must have a state labeled $\{q_1, q_2\}$ and a transition

$$\delta(\{q_0\}, a) = \{q_1, q_2\}.$$

In state q_0, the nfa has no specified transition when the input is b; therefore,

$$\delta(\{q_0\}, b) = \varnothing.$$

A state labeled $\varnothing$ represents an impossible move for the nfa and, therefore, means nonacceptance of the string. Consequently, this state in the dfa must be a nonfinal trap state.

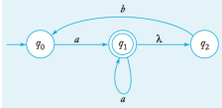


FIGURE 2.12

Equivalence of NFAs and DFAs [L-ToC-17]

We have now introduced into the dfa the state $\{q_1, q_2\}$, so we need to find the transitions out of this state. Remember that this state of the dfa corresponds to two possible states of the nfa, so we must refer back to the nfa. If the nfa is in state q_1 and reads an a, it can go to q_1. Furthermore, from q_1 the nfa can make a λ-transition to q_2. If, for the same input, the nfa is in state q_2, then there is no specified transition. Therefore,

$$\delta(\{q_1, q_2\}, a) = \{q_1, q_2\}.$$

Similarly,

$$\delta(\{q_1, q_2\}, b) = \{q_0\}.$$

At this point, every state has all transitions defined. The result, shown in Figure 2.13, is a dfa, equivalent to the nfa with which we started. The nfa in Figure 2.12 accepts any string for which $\delta^*(q_0, w)$ contains q_1. For the corresponding dfa to accept every such w, any state whose label includes q_1 must be made a final state.

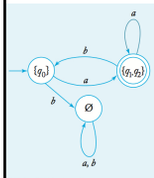


FIGURE 2.13

NFA Application Example: Text Search [HMU-ToC-07]

Example 2.14: Suppose we want to design an NFA to recognize occurrences of the words **web** and **ebay**. The transition diagram for the NFA designed using the rules above is in Fig. 2.16. State 1 is the start state, and we use Σ to stand for the set of all printable ASCII characters. States 2 through 4 have the job of recognizing **web**, while states 5 through 8 recognize **ebay**. □

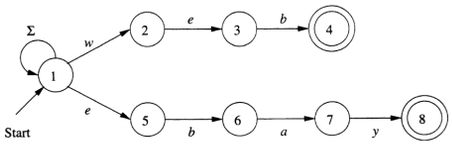


Figure 2.16: An NFA that searches for the words **web** and **ebay**

DFA Application Example: Text Search [HMU-ToC-07]

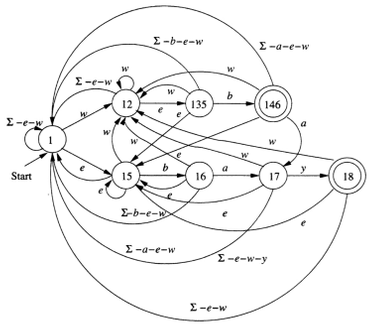


Figure 2.17: Conversion of the NFA from Fig. 2.16 to a DFA

NFA Application Example: Uses of ε-Transition [HMU-ToC-07]

Example 2.16: In Fig. 2.18 is an ϵ-NFA that accepts decimal numbers consisting of:

1. An optional + or − sign,
2. A string of digits,
3. A decimal point, and
4. Another string of digits. Either this string of digits, or the string (2) can be empty, but at least one of the two strings of digits must be nonempty.

Of particular interest is the transition from q_0 to q_1 on any of ϵ, +, or −. Thus, state q_1 represents the situation in which we have seen the sign if there is one, and perhaps some digits, but not the decimal point. State q_2 represents the situation where we have just seen the decimal point, and may or may not have seen prior digits. In q_4 we have definitely seen at least one digit, but not the decimal point. Thus, the interpretation of q_3 is that we have seen a

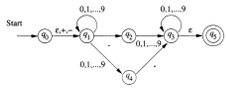


Figure 2.18: An ϵ-NFA accepting decimal numbers

decimal point and at least one digit, either before or after the decimal point. We may stay in q_3 reading whatever digits there are, and also have the option of "guessing" the string of digits is complete and going spontaneously to q_5, the accepting state. □

DFA Application Example: Eliminating ε-Transition [HMU-ToC-07]

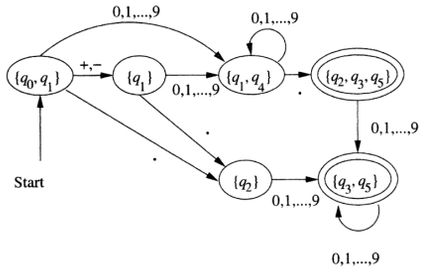


Figure 2.22: The DFA D that eliminates ϵ-transitions from Fig. 2.18

NFA Application Example: Uses of ε-Transition [HMU-ToC-07]

Example 2.17: The strategy we outlined in Example 2.14 for building an NFA that recognizes a set of keywords can be simplified further if we allow ϵ-transitions. For instance, the NFA recognizing the keywords **web** and **ebay**, which we saw in Fig. 2.16, can also be implemented with ϵ-transitions as in Fig. 2.19. In general, we construct a complete sequence of states for each keyword, as if it were the only word the automaton needed to recognize. Then, we add a new start state (state 9 in Fig. 2.19), with ϵ-transitions to the start-states of the automata for each of the keywords. □

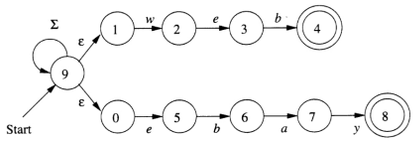


Figure 2.19: Using ϵ-transitions to help recognize keywords

Regular Expressions

♣ **Regular expressions**

- **We can use the regular operations to build up expressions describing languages, which are called *regular expressions*.**
- **The value of a regular expression is a language.**

♣ **Examples**

- **The value of the regular expression (0 ∪ 1)0* is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s.**
- **The value of the regular expression (0 ∪ 1)* is the language consisting of all possible strings of 0s and 1s.**

Regular Expressions: Formal Definition [S-ToC-13]

♣ **Regular Expressions**

DEFINITION 1.52

Say that R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

In items 1 and 2, the regular expressions a and ε represent the languages $\{a\}$ and $\{\varepsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

Regular Expressions: Formal Definition [S-ToC-13]

♣ **Notes**

- **Do not confuse the regular expressions ε and ∅. The expression ε represents the language containing a single string, the empty string ε; whereas ∅ (the empty set) represents the empty language that doesn't contain any strings.**
- **R_1 and R_2 always are smaller than R. Thus we actually are defining regular expressions in terms of smaller regular expressions and thereby avoiding circularity. A definition of this type is called an *inductive definition*.**
- **Parentheses in an expression may be omitted. If they are, evaluation is done in the precedence order: star, then concatenation, then union.**
- **$R^* = R^+ \cup \varepsilon$. In addition, we let R^k be shorthand for the concatenation of k R's strings with each other.**

Regular Expressions: Formal Definition [L-ToC-17]

♣ **Regular Expressions**

DEFINITION 3.1

Let Σ be a given alphabet. Then

1. $\varnothing$, λ, and $a \in \Sigma$ are all regular expressions. These are called **primitive regular expressions**.
2. If r_1 and r_2 are regular expressions, so are $r_1 + r_2$, $r_1 \cdot r_2$, r_1^*, and (r_1).
3. A string is a regular expression if and only if it can be derived from the primitive regular expressions by a finite number of applications of the rules in (2).

Regular Expressions: Formal Definition [L-ToC-17]

♣ **Languages Associated with Regular Expressions**

DEFINITION 3.2

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

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

If r_1 and r_2 are regular expressions, then

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)\,L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

Regular Expression: Examples [S-ToC-13]

In the following instances, we assume that the alphabet Σ is $\{0,1\}$.

1. $0^*10^* = \{w \mid w$ contains a single $1\}$.
2. $\Sigma^*1\Sigma^* = \{w \mid w$ has at least one $1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w$ contains the string 001 as a substring$\}$.
4. $1^*(01^*)^* = \{w \mid$ every 0 in w is followed by at least one $1\}$.
5. $(\Sigma\Sigma)^* = \{w \mid w$ is a string of even length$\}$.[5]
6. $(\Sigma\Sigma\Sigma)^* = \{w \mid$ the length of w is a multiple of $3\}$.
7. $01 \cup 10 = \{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w$ starts and ends with the same symbol$\}$.
9. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.
   The expression $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$, so the concatenation operation adds either 0 or ε before every string in 1^*.
10. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
11. $1^*\emptyset = \emptyset$.
    Concatenating the empty set to any set yields the empty set.
12. $\emptyset^* = \{\varepsilon\}$.
    The star operation puts together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, giving only the empty string.

Regular Expressions: Exercises

- $R \cup \varnothing = ?$ **(*R*)**
  (Adding the empty language to any other language will not change it)
- $R \circ \varepsilon = ?$ **(*R*)**
  (Joining the empty string to any string will not change it)
- $R \cup \varepsilon = ?$ **(Not necessarily *R*)**
  (Ex: if *R* = 0, then *L*(*R*) = {0} but *L*(*R* ∪ ε) = {0, ε})
- $R \circ \varnothing = ?$ **(Not necessarily *R*)**
  (Ex: if *R* = 0, then *L*(*R*) = {0} but *L*(*R* ∘ ∅) = ∅)

Equivalence of Regular Expressions and FAs [S-ToC-13]

♣ **Equivalence theorem**

THEOREM 1.54

A language is regular if and only if some regular expression describes it.

LEMMA 1.55

If a language is described by a regular expression, then it is regular.

LEMMA 1.60

If a language is regular, then it is described by a regular expression.

Converting a Regular Expressions to an NFA [S-ToC-13]

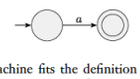
LEMMA 1.55

If a language is described by a regular expression, then it is regular.

PROOF IDEA Say that we have a regular expression R describing some language A. We show how to convert R into an NFA recognizing A. By Corollary 1.40, if an NFA recognizes A then A is regular.

PROOF Let's convert R into an NFA N. We consider the six cases in the formal definition of regular expressions.

1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.



Note that this machine fits the definition of an NFA but not that of a DFA because it has some states with no exiting arrow for each possible input symbol. Of course, we could have presented an equivalent DFA here; but an NFA is all we need for now, and it is easier to describe.

Formally, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.

Converting a Regular Expressions to an NFA [S-ToC-13]

2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$, and the following NFA recognizes $L(R)$.

Formally, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, where $\delta(r, b) = \emptyset$ for any r and b.

3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$.

Formally, $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, where $\delta(r, b) = \emptyset$ for any r and b.

4. $R = R_1 \cup R_2$.
5. $R = R_1 \circ R_2$.
6. $R = R_1^*$.

For the last three cases, we use the constructions given in the proofs that the class of regular languages is closed under the regular operations. In other words, we construct the NFA for R from the NFAs for R_1 and R_2 (or just R_1 in case 6) and the appropriate closure construction.

Converting a Regular Expressions to an NFA [S-ToC-13]

EXAMPLE 1.56

We convert the regular expression $(ab \cup a)^*$ to an NFA in a sequence of stages. We build up from the smallest subexpressions to larger subexpressions until we have an NFA for the original expression, as shown in the following diagram. Note that this procedure generally doesn't give the NFA with the fewest states. In this example, the procedure gives an NFA with eight states, but the smallest equivalent NFA has only two states. Can you find it?

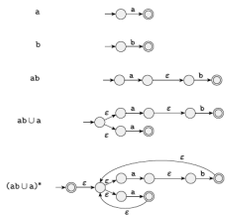


FIGURE 1.57
Building an NFA from the regular expression $(ab \cup a)^*$

Converting a Regular Expressions to an NFA [S-ToC-13]

EXAMPLE 1.58

In Figure 1.59, we convert the regular expression $(a \cup b)^*aba$ to an NFA. A few of the minor steps are not shown.

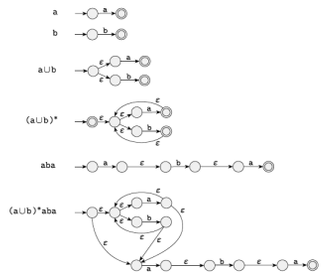


FIGURE 1.59
Building an NFA from the regular expression $(a \cup b)^*aba$

Equivalence of Regular Expressions and FAs

♣Converting a DFA to a regular expression

- **The procedure of converting DFAs into equivalent regular expressions can be breaked into two parts, using a new type of finite automaton called a *generalized nondeterministic finite automaton (GNFA)*.**
- **First we convert DFAs into GNFAs, and then GNFAs into regular expressions.**

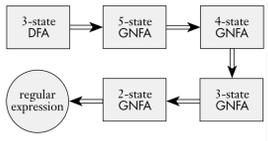


FIGURE 1.62
Typical stages in converting a DFA to a regular expression

Generalized Non-deterministic Finite Automata (GNFAs): An Example [S-ToC-13]

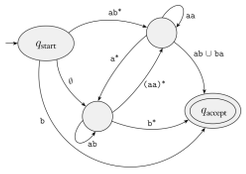


FIGURE 1.61
A generalized nondeterministic finite automaton

For convenience, we require that GNFAs always have a special form that meets the following conditions.

- The start state has transition arrows going to every other state but no arrows coming in from any other state.
- There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
- Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.

Generalized Non-deterministic Finite Automata (GNFAs): Formal Definition [S-ToC-13]

DEFINITION 1.64

A ***generalized nondeterministic finite automaton*** is a 5-tuple, $(Q, \Sigma, \delta, q_{start}, q_{accept})$, where

1. Q is the finite set of states,
2. Σ is the input alphabet,
3. $\delta: (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \longrightarrow \mathcal{R}$ is the transition function,
4. q_{start} is the start state, and
5. q_{accept} is the accept state.

A GNFA accepts a string w in Σ^* if $w = w_1 w_2 \cdots w_k$, where each w_i is in Σ^* and a sequence of states $q_0, q_1, \ldots, q_k$ exists such that

1. $q_0 = q_{start}$ is the start state,
2. $q_k = q_{accept}$ is the accept state, and
3. for each i, we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$; in other words, R_i is the expression on the arrow from q_{i-1} to q_i.

Algorithm for Converting GNFAs into Regular Expressions [S-ToC-13]

♣Algorithm for converting a GNFAs into regular expressions

CONVERT(G):

1. Let k be the number of states of G.
2. If $k = 2$, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R. Return the expression R.
3. If $k > 2$, we select any state $q_{rip} \in Q$ different from q_{start} and q_{accept} and let G' be the GNFA $(Q', \Sigma, \delta', q_{start}, q_{accept})$, where
$$Q' = Q - \{q_{rip}\},$$
and for any $q_i \in Q' - \{q_{accept}\}$ and any $q_j \in Q' - \{q_{start}\}$, let
$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4),$$
for $R_1 = \delta(q_i, q_{rip})$, $R_2 = \delta(q_{rip}, q_{rip})$, $R_3 = \delta(q_{rip}, q_j)$, and $R_4 = \delta(q_i, q_j)$.
4. Compute CONVERT(G') and return this value.

♣The correctness of the algorithm

CLAIM 1.65

For any GNFA G, CONVERT(G) is equivalent to G.

Converting DFAs into Regular Expressions: Examples [S-ToC-13]

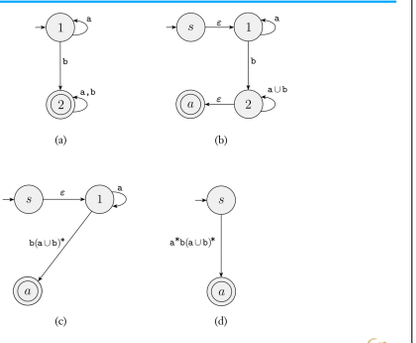


FIGURE 1.67
Converting a two-state DFA to an equivalent regular expression

Converting DFAs into Regular Expressions: Examples [S-ToC-13]

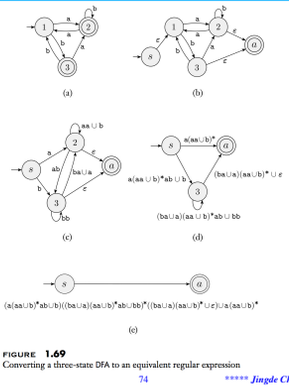


FIGURE 1.69
Converting a three-state DFA to an equivalent regular expression

Regular Expression Examples [L-ToC-17]

EXAMPLE 3.1

For $\Sigma = \{a, b, c\}$, the string

$$(a + b \cdot c)^* \cdot (c + \varnothing)$$

is a regular expression, since it is constructed by application of the above rules. For example, if we take $r_1 = c$ and $r_2 = \varnothing$, we find that $c + \varnothing$ and $(c + \varnothing)$ are also regular expressions. Repeating this, we eventually generate the whole string. On the other hand, $(a + b +)$ is not a regular expression, since there is no way it can be constructed from the primitive regular expressions.

Regular Expression Examples [L-ToC-17]

EXAMPLE 3.2

Exhibit the language $L(a^* \cdot (a + b))$ in set notation.

$$\begin{aligned} L(a^* \cdot (a + b)) &= L(a^*) L(a + b) \\ &= (L(a))^* (L(a) \cup L(b)) \\ &= \{\lambda, a, aa, aaa, ...\} \{a, b\} \\ &= \{a, aa, aaa, ..., b, ab, aab, ...\}. \end{aligned}$$

EXAMPLE 3.3

For $\Sigma = \{a, b\}$, the expression

$$r = (a + b)^* (a + bb)$$

is regular. It denotes the language

$$L(r) = \{a, bb, aa, abb, ba, bbb, ...\}.$$

We can see this by considering the various parts of r. The first part, $(a + b)^*$, stands for any string of a's and b's. The second part, $(a + bb)$ represents either an a or a double b. Consequently, $L(r)$ is the set of all strings on $\{a, b\}$, terminated by either an a or a bb.

Regular Expression Examples [L-ToC-17]

EXAMPLE 3.4

The expression

$$r = (aa)^* (bb)^* b$$

denotes the set of all strings with an even number of a's followed by an odd number of b's; that is,

$$L(r) = \{a^{2n} b^{2m+1} : n \geq 0,\ m \geq 0\}.$$

EXAMPLE 3.5

For $\Sigma = \{0, 1\}$, give a regular expression r such that

$$L(r) = \{w \in \Sigma^* : w \text{ has at least one pair of consecutive zeros}\}.$$

One can arrive at an answer by reasoning something like this: Every string in $L(r)$ must contain 00 somewhere, but what comes before and what goes after is completely arbitrary. An arbitrary string on $\{0, 1\}$ can be denoted by $(0 + 1)^*$. Putting these observations together, we arrive at the solution

$$r = (0 + 1)^* 00 (0 + 1)^*.$$

Regular Expression Examples [L-ToC-17]

EXAMPLE 3.6

Find a regular expression for the language

$$L = \{w \in \{0, 1\}^* : w \text{ has no pair of consecutive zeros}\}.$$

Even though this looks similar to Example 3.5, the answer is harder to construct. One helpful observation is that whenever a 0 occurs, it must be followed immediately by a 1. Such a substring may be preceded and followed by an arbitrary number of 1's. This suggests that the answer involves the repetition of strings of the form $1 \cdots 101 \cdots 1$, that is, the language denoted by the regular expression $(1^*011^*)^*$. However, the answer is still incomplete, since the strings ending in 0 or consisting of all 1's are unaccounted for. After taking care of these special cases we arrive at the answer

$$r = (1^*011^*)^* (0 + \lambda) + 1^* (0 + \lambda).$$

If we reason slightly differently, we might come up with another answer. If we see L as the repetition of the strings 1 and 01, the shorter expression

$$r = (1 + 01)^* (0 + \lambda)$$

might be reached. Although the two expressions look different, both answers are correct, as they denote the same language. Generally, there are an unlimited number of regular expressions for any given language.

Note that this language is the complement of the language in Example 3.5. However, the regular expressions are not very similar and do not suggest clearly the close relationship between the languages.

Regular Expressions and Regular Languages [L-ToC-17]

♣ **Regular expressions denote regular languages**

THEOREM 3.1

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

♣ **Regular languages denote regular expressions**

THEOREM 3.2

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

Regular Expression Examples [L-ToC-17]

EXAMPLE 3.7

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

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

Automata for $(a + bb)$ and $(ba^* + \lambda)$, constructed directly from first principles, are given in Figure 3.6. Putting these together using the construction in Theorem 3.1, we get the solution in Figure 3.7.

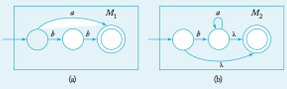


FIGURE 3.6 (a) M_1 accepts $L(a + bb)$. (b) M_2 accepts $L(ba^* + \lambda)$.

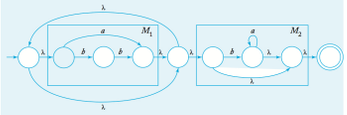


FIGURE 3.7 Automaton accepts $L((a + bb)^* (ba^* + \lambda))$.

Regular Expressions and Regular Languages

♣ **Generalized transition graph**

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

Regular Expression Examples [L-ToC-17]

EXAMPLE 3.8

Figure 3.8 represents a generalized transition graph. The language accepted by it is $L(a^* + a^*(a + b)c^*)$, as should be clear from an inspection of the graph. The edge (q_0, q_0) labeled a is a cycle that can generate any number of a's, that is, it represents $L(a^*)$. We could have labeled this edge a^* without changing the language accepted by the graph.

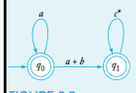


FIGURE 3.8

EXAMPLE 3.9

The GTG in Figure 3.9(a) is not complete. Figure 3.9(b) shows how it is completed.

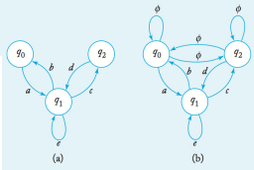


FIGURE 3.9

Regular Expression Examples [L-ToC-17]

EXAMPLE 3.10

Consider the complete GTG in Figure 3.11. To remove q_2, we first introduce some new edges. We

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

When this is done, we remove q_2 and all associated edges. This gives the GTG in Figure 3.12. You can explore the equivalence of the two GTGs by seeing how regular expressions such as af^*c and c^*ab are generated.

FIGURE 3.11

FIGURE 3.12

Regular Expression Examples [L-ToC-17]

EXAMPLE 3.11

Find a regular expression for the language

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

An attempt to construct a regular expression directly from this description leads to all kinds of difficulties. On the other hand, finding an nfa for it is easy as long as we use vertex labeling effectively. We label the vertices with EE to denote an even number of a's and b's, with OE to denote an odd number of a's and an even number of b's, and so on. With this we easily get the solution that, after conversion into a complete generalized transition graph, is in Figure 3.13.

We now apply the conversion to a regular expression, using procedure *nfa-to-rex*. To remove the state OE, we apply Equation (3.3). The edge between EE and itself will have the label

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

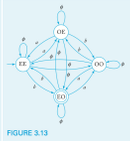


FIGURE 3.13

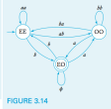


FIGURE 3.14

FIGURE 3.15

We continue in this manner until we get the GTG in Figure 3.14. Next, the state OO is removed, which gives Figure 3.15. Finally, we get the correct regular expression from Equation (3.2).

Regular Grammars

♣ **Grammars and languages**

- To study languages mathematically, we need a mechanism to describe them.
- Grammars are often an alternative way of specifying languages.
- Whenever we define a language family through an automaton or in some other way, we are interested in knowing what kind of grammar we can associate with the family.

♣ **Regular grammars** 正则文法

- A third way of describing regular languages is by means of certain ***regular grammars***.
- Regular grammars generate regular languages.

正则文法生成正则语言

Grammars [L-ToC-17]

♣ **Formal definition of grammars**

DEFINITION 1.1

A grammar G is defined as a quadruple

$$G = (V, T, S, P),$$

where V is a finite set of objects called **variables**,
T is a finite set of objects called **terminal symbols**,
$S \in V$ is a special symbol called the **start** variable,
P is a finite set of **productions**.

It will be assumed without further mention that the sets V and T are non-empty and disjoint.

全部是有穷的
V:变量集
T:结束符号集
S∈V:起始变量集
P:衍生规则集

♣ **Languages generated by grammars**

DEFINITION 1.2

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

$$L(G) = \left\{ w \in T^* : S \overset{*}{\Rightarrow} w \right\}$$

is the language generated by G.

Grammars

♣ **Production rules** 衍生规则

- The ***production rules*** are the heart of a grammar; they specify how the grammar transforms one string into another, and through this they define a language associated with the grammar.
- All production rules are of the form $x \rightarrow y$, where x is an element of $(V \cup T)^+$ and y is in $(V \cup T)^*$.

♣ **Applications of production rules**

- The productions are applied in the following manner: Given a string w of the form $w = uxv$, we say the production $x \rightarrow y$ is applicable to this string, and we may use it to replace x with y, thereby obtaining a new string $z = uyv$.
- This is written as $w \Rightarrow z$. We say that w derives z or that z is derived from w.

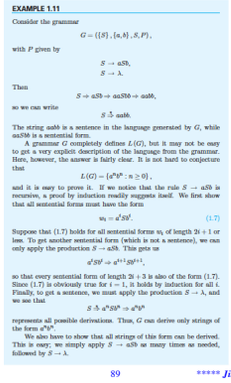
单步获得: $w \Rightarrow z$
多步获得: $w \overset{*}{\Rightarrow} z$
(可以为0步)

Grammars

♣ **Applications of production rules**

- Successive strings are derived by applying the productions of the grammar in arbitrary order.
- A production can be used whenever it is applicable, and it can be applied as often as desired.
- If $w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$, we say that w_1 derives w_n and write $w_1 \Rightarrow^* w_n$. The * indicates that an unspecified number of steps (including zero) can be taken to derive w_n from w_1.
- By applying the production rules in a different order, a given grammar can normally generate many strings.
- The set of all such terminal strings is the language defined or generated by the grammar.

Grammar Examples [L-ToC-17]



EXAMPLE 1.11

Consider the grammar

$$G = (\{S\}, \{a, b\}, S, P),$$

with P given by

$$S \rightarrow aSb,$$
$$S \rightarrow \lambda.$$

Then

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb,$$

so we can write

$$S \overset{*}{\Rightarrow} aabb.$$

The string $aabb$ is a sentence in the language generated by G, while $aaSbb$ is a sentential form.

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

$$L(G) = \{a^n b^n : n \geq 0\},$$

and it is easy to prove it. If we notice that the rule $S \rightarrow aSb$ is recursive, a proof by induction readily suggests itself. We first show that all sentential forms must have the form

$$w_i = a^i S b^i. \tag{1.7}$$

Suppose that (1.7) holds for all sentential forms w_i of length $2i + 1$ or less. To get another sentential form (which is not a sentence), we can only apply the production $S \rightarrow aSb$. This gets us

$$a^i S b^i \Rightarrow a^{i+1} S b^{i+1},$$

so that every sentential form of length $2i + 3$ is also of the form (1.7). Since (1.7) is obviously true for $i = 1$, it holds by induction for all i. Finally, to get a sentence, we must apply the production $S \rightarrow \lambda$, and we see that

$$S \overset{*}{\Rightarrow} a^n S b^n \Rightarrow a^n b^n$$

represents all possible derivations. Thus, G can derive only strings of the form $a^n b^n$.

We also have to show that all strings of this form can be derived. This is easy; we simply apply $S \rightarrow aSb$ as many times as needed, followed by $S \rightarrow \lambda$.

Grammar Examples [L-ToC-17]

EXAMPLE 1.12

Find a grammar that generates

$$L = \{a^n b^{n+1} : n \geq 0\}.$$

The idea behind the previous example can be extended to this case. All we need to do is generate an extra b. This can be done with a production $S \rightarrow Ab$, with other productions chosen so that A can derive the language in the previous example. Reasoning in this fashion, we get the grammar $G = (\{S, A\}, \{a, b\}, S, P)$, with productions

$$S \rightarrow Ab,$$
$$A \rightarrow aAb,$$
$$A \rightarrow \lambda.$$

Derive a few specific sentences to convince yourself that this works.

Regular Grammars [L-ToC-17]

♣ **Formal definition of regular grammars**

DEFINITION 3.3

右线性

A grammar $G = (V, T, S, P)$ is said to be **right-linear** if all productions are of the form

$$A \to xB,$$
$$A \to x,$$

where $A, B \in V$, and $x \in T^*$. A grammar is said to be **left-linear** if all productions are of the form

左线性

$$A \to Bx,$$

or

$$A \to x.$$

A **regular grammar** is one that is either right-linear or left-linear.

♣ **Notes**

- **In a regular grammar, at most one variable appears on the right side of any production.**
- **Furthermore, that variable must consistently be either the rightmost or leftmost symbol of the right side of any production.**

Regular Grammars and Regular Languages [L-ToC-17]

♣ **Right-linear grammars generate regular languages** 右线性文法产生正则语言

THEOREM 3.3

Let $G = (V, T, S, P)$ be a right-linear grammar. Then $L(G)$ is a regular language.

♣ **Right-linear grammars for regular languages**

THEOREM 3.4

If L is a regular language on the alphabet Σ, then there exists a right-linear grammar $G = (V, \Sigma, S, P)$ such that $L = L(G)$.

♣ **Equivalence**

THEOREM 3.5

语言L是正则的 <=> 有左线性文法G使得 L=L(G)

A language L is regular if and only if there exists a left-linear grammar G such that $L = L(G)$.

THEOREM 3.6

A language L is regular if and only if there exists a regular grammar G such that $L = L(G)$.

语言L是正则的 <=> 有正则文法G使得 L=L(G)

Regular Languages, Expressions, and Grammars [L-ToC-17]

We now have several ways of describing regular languages: dfa's, nfa's, regular expressions, and regular grammars. While in some instances one or the other of these may be most suitable, they are all equally powerful.

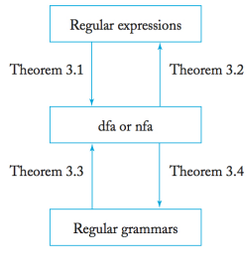


FIGURE 3.19

正则语言的封闭性

Closure Properties of Regular Languages [L-ToC-17]

♣ **Closure under simple set operations**

THEOREM 4.1

If L_1 and L_2 are regular languages, then so are $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 L_2$, $\overline{L_1}$, and L_1^*. We say that the family of regular languages is closed under union, intersection, concatenation, complementation, and star-closure.

在简单的集合操作下是封闭的

THEOREM 4.2

The family of regular languages is closed under reversal.

对反转操作封闭

Closure Properties of Regular Languages [L-ToC-17]

♣ **Closure under other operations**

DEFINITION 4.1

Suppose Σ and Γ are alphabets. Then a function

$$h : \Sigma \to \Gamma^*$$

is called a **homomorphism**. In words, a homomorphism is a substitution in which a single letter is replaced with a string. The domain of the function h is extended to strings in an obvious fashion; if

$$w = a_1 a_2 \cdots a_n,$$

then

$$h(w) = h(a_1)\, h(a_2) \cdots h(a_n).$$

If L is a language on Σ, then its **homomorphic image** is defined as

$$h(L) = \{h(w) : w \in L\}.$$

Closure Properties of Regular Languages [L-ToC-17]

♣ **Closure under other operations**

THEOREM 4.3

Let h be a homomorphism. If L is a regular language, then its homomorphic image $h(L)$ is also regular. The family of regular languages is therefore closed under arbitrary homomorphisms.

对同构操作封闭

DEFINITION 4.2

Let L_1 and L_2 be languages on the same alphabet. Then the **right quotient** of L_1 with L_2 is defined as

$$L_1/L_2 = \{x : xy \in L_1 \text{ for some } y \in L_2\}. \quad (4.1)$$

THEOREM 4.4

If L_1 and L_2 are regular languages, then L_1/L_2 is also regular. We say that the family of regular languages is closed under right quotient with a regular language.

对右商封闭

A Summary of the Closure Properties of Regular Languages [HMU-ToC-07]

1. The union of two regular languages is regular.
2. The intersection of two regular languages is regular.
3. The complement of a regular language is regular.
4. The difference of two regular languages is regular.
5. The reversal of a regular language is regular.
6. The closure (star) of a regular language is regular.
7. The concatenation of regular languages is regular.
8. A homomorphism (substitution of strings for symbols) of a regular language is regular.
9. The inverse homomorphism of a regular language is regular.

Elementary Properties about Regular Languages

♣ **Elementary question**

- **A 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.**

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.

THEOREM 4.6

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

THEOREM 4.7

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

A Summary of the Algorithmic Problems Related to Regular Languages [LP-ToC-98]

Theorem 2.6.1: *(a) There is an exponential algorithm which, given a nondeterministic finite automaton, constructs an equivalent deterministic finite automaton.*
(b) There is a polynomial algorithm which, given a regular expression, constructs an equivalent nondeterministic finite automaton.
(c) There is an exponential algorithm which, given a nondeterministic finite automaton, constructs an equivalent regular expression.
(d) There is a polynomial algorithm which, given a deterministic finite automaton, constructs an equivalent deterministic finite automaton with the smallest possible number of states.
(e) There is a polynomial algorithm which, given two deterministic finite automata, decides whether they are equivalent.
(f) There is an exponential algorithm which, given two nondeterministic finite automata, decides whether they are equivalent; similarly for the equivalence of two regular expressions.

非正则语言

Non-regular Languages

♣ **An example language over Σ = {0,1}**

- **B = { 0^n1^n | $n \geq 0$ }.**
- **If we attempt to find a DFA that recognizes B, we discover that the machine seems to need to remember how many 0s have been seen so far as it reads the input.**
- **Because the number of 0s is not limited, the machine will have to keep track of an unlimited number of possibilities. But it cannot do so with any finite number of states.**

n不是有限的数，状态数不确定，故无法找到DFA

♣ **Other two example languages over Σ = {0,1}**

- **C = { w | w has an equal number of 0s and 1s }**
- **D = { w | w has an equal number of occurrences of 01 and 10 as substrings }.**
- **C is not regular, but surprisingly D is regular!**

The Pumping Lemma for Regular Languages [S-ToC-13]

♣ **A necessary property of regular languages** 正则语言的必要性质

- **[D] All strings in a regular language can be "pumped" if they are at least as long as a certain special value, called the *pumping length*.**
- **That means each such string contains a section that can be repeated any number of times with the resulting string remaining in the language.**

正则语言必须能构造出这个引理

♣ **The pumping lemma**

THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Identifying Non-regular Languages

♣ **The idea**

- **To use the pumping lemma to prove that a language B is not regular, first assume that B is regular and should satisfy the pumping lemma, then obtain a contradiction to show the assumption does not hold.**

♣ **The steps**

- **Use the pumping lemma to guarantee the existence of a pumping length p such that all strings of length p or greater in B can be pumped.**
- **Find a string s in B that has length p or greater but that cannot be pumped.**
- **Demonstrate that s cannot be pumped by considering all ways of dividing s into x, y, and z (taking condition 3 of the pumping lemma into account if convenient) and, for each such division, finding a value i where $xy^iz \notin$ B.**
- **The existence of s contradicts the pumping lemma if B were regular. Hence B cannot be regular.**

Non-regular Languages: An Example [S-ToC-13]

EXAMPLE 1.73

Let B be the language $\{0^n1^n | n \geq 0\}$. We use the pumping lemma to prove that B is not regular. The proof is by contradiction.

Assume to the contrary that B is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string 0^p1^p. Because s is a member of B and s has length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in B. We consider three cases to show that this result is impossible.

1. The string y consists only of 0s. In this case, the string $xyyz$ has more 0s than 1s and so is not a member of B, violating condition 1 of the pumping lemma. This case is a contradiction.
2. The string y consists only of 1s. This case also gives a contradiction.
3. The string y consists of both 0s and 1s. In this case, the string $xyyz$ may have the same number of 0s and 1s, but they will be out of order with some 1s before 0s. Hence it is not a member of B, which is a contradiction.

Thus a contradiction is unavoidable if we make the assumption that B is regular, so B is not regular. Note that we can simplify this argument by applying condition 3 of the pumping lemma to eliminate cases 2 and 3.

In this example, finding the string s was easy because any string in B of length p or more would work. In the next two examples, some choices for s do not work so additional care is required. ■

Non-regular Languages: An Example [S-ToC-13]

EXAMPLE 1.75

Let $F = \{ww | w \in \{0,1\}^*\}$. We show that F is nonregular, using the pumping lemma.

Assume to the contrary that F is regular. Let p be the pumping length given by the pumping lemma. Let s be the string 0^p10^p1. Because s is a member of F and s has length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the three conditions of the lemma. We show that this outcome is impossible.

Condition 3 is once again crucial because without it we could pump s if we let x and z be the empty string. With condition 3 the proof follows because y must consist only of 0s, so $xyyz \notin F$.

Observe that we chose $s = 0^p10^p1$ to be a string that exhibits the "essence" of the nonregularity of F, as opposed to, say, the string 0^p0^p. Even though 0^p0^p is a member of F, it fails to demonstrate a contradiction because it can be pumped.

Identifying Non-regular Languages [L-ToC-17]

EXAMPLE 4.6

Is the language $L = \{a^nb^n : n \geq 0\}$ regular? The answer is no, as we show using a proof by contradiction.

Suppose L is regular. Then some dfa $M = (Q, \{a,b\}, \delta, q_0, F)$ exists for it. Now look at $\delta^*(q_0, a^i)$ for $i = 1, 2, 3, \ldots$. Since there are an unlimited number of i's, but only a finite number of states in M, the pigeonhole principle tells us that there must be some state, say q, such that

$$\delta^*(q_0, a^n) = q$$

and

$$\delta^*(q_0, a^m) = q,$$

with $n \neq m$. But since M accepts a^nb^n we must have

$$\delta^*(q, b^n) = q_f \in F.$$

From this we can conclude that

$$\begin{aligned}\delta^*(q_0, a^mb^n) &= \delta^*(\delta^*(q_0, a^m), b^n) \\ &= \delta^*(q, b^n) \\ &= q_f.\end{aligned}$$

This contradicts the original assumption that M accepts a^mb^n only if $n = m$, and leads us to conclude that L cannot be regular.

The Pumping Lemma for Regular Languages [L-ToC-17]

THEOREM 4.8

Let L be an infinite regular language. Then there exists some positive integer m such that any $w \in L$ with $|w| \geq m$ can be decomposed as

$$w = xyz$$

with

$$|xy| \leq m,$$

and

$$|y| \geq 1,$$

such that

$$w_i = xy^iz, \qquad (4.2)$$

is also in L for all $i = 0, 1, 2, \ldots$.

To paraphrase this, every sufficiently long string in L can be broken into three parts in such a way that an arbitrary number of repetitions of the middle part yields another string in L. We say that the middle string is "pumped," hence the term pumping lemma for this result.

Identifying Non-regular Languages [L-ToC-17]

EXAMPLE 4.7

Use the pumping lemma to show that $L = \{a^nb^n : n \geq 0\}$ is not regular. Assume that L is regular, so that the pumping lemma must hold. We do not know the value of m, but whatever it is, we can always choose $n = m$. Therefore, the substring y must consist entirely of a's. Suppose $|y| = k$. Then the string obtained by using $i = 0$ in Equation (4.2) is

$$w_0 = a^{m-k}b^m$$

and is clearly not in L. This contradicts the pumping lemma and thereby indicates that the assumption that L is regular must be false.

Non-regular Language Examples [L-ToC-17]

EXAMPLE 4.9

Let $\Sigma = \{a, b\}$. The language

$$L = \{w \in \Sigma^* : n_a(w) < n_b(w)\}$$

is not regular.

Suppose we are given m. Since we have complete freedom in choosing w, we pick $w = a^mb^{m+1}$. Now, because $|xy|$ cannot be greater than m, the opponent cannot do anything but pick a y with all a's, that is

$$y = a^k, \qquad 1 \leq k \leq m.$$

We now pump up, using $i = 2$. The resulting string

$$w_2 = a^{m+k}b^{m+1}$$

is not in L. Therefore, the pumping lemma is violated, and L is not regular.

Non-regular Language Examples [L-ToC-17]

EXAMPLE 4.10

The language

$$L = \left\{ (ab)^n a^k : n > k, k \geq 0 \right\}$$

is not regular.

Given m, we pick as our string

$$w = (ab)^{m+1} a^m,$$

which is in L. Because of the constraint $|xy| \leq m$, both x and y must be in the part of the string made up of ab's. The choice of x does not affect the argument, so let us see what can be done with y. If our opponent picks $y = a$, we choose $i = 0$ and get a string not in $L\left((ab)^* a^*\right)$. If the opponent picks $y = ab$, we can choose $i = 0$ again. Now we get the string $(ab)^m a^m$, which is not in L. In the same way, we can deal with any possible choice by the opponent, thereby proving our claim.

Non-regular Language Examples [L-ToC-17]

EXAMPLE 4.11

Show that

$$L = \{a^n : n \text{ is a perfect square}\}$$

is not regular.

Given the opponent's choice of m, we pick

$$w = a^{m^2}.$$

If $w = xyz$ is the decomposition, then clearly

$$y = a^k$$

with $1 \leq k \leq m$. In that case,

$$w_0 = a^{m^2 - k}$$

But $m^2 - k > (m-1)^2$, so that w_0 cannot be in L. Therefore, the language is not regular.

Non-regular Language Examples [L-ToC-17]

EXAMPLE 4.12

Show that the language

$$L = \left\{ a^n b^k c^{n+k} : n \geq 0, k \geq 0 \right\}$$

is not regular.

It is not difficult to apply the pumping lemma directly, but it is even easier to use closure under homomorphism. Take

$$h(a) = a, h(b) = a, h(c) = c,$$

then

$$\begin{aligned} h(L) &= \left\{ a^{n+k} c^{n+k} : n + k \geq 0 \right\} \\ &= \left\{ a^i c^i : i \geq 0 \right\}. \end{aligned}$$

But we know this language is not regular; therefore, L cannot be regular either.

Non-regular Language Examples [L-ToC-17]

EXAMPLE 4.13

Show that the language

$$L = \left\{ a^n b^l : n \neq l \right\}$$

is not regular.

Here we need a bit of ingenuity to apply the pumping lemma directly. Choosing a string with $n = l + 1$ or $n = l + 2$ will not do, since our opponent can always choose a decomposition that will make it impossible to pump the string out of the language (that is, pump it so that it has an equal number of a's and b's). We must be more inventive. Let us take $n = m!$ and $l = (m+1)!$. If the opponent now chooses a y (by necessity consisting of all a's) of length $k < m$, we pump i times to generate a string with $m! + (i-1)\,k$ a's. We can get a contradiction of the pumping lemma if we can pick i such that

$$m! + (i-1)\,k = (m+1)!$$

This is always possible since

$$i = 1 + \frac{m\,m!}{k}$$

and $k \leq m$. The right side is therefore an integer, and we have succeeded in violating the conditions of the pumping lemma.

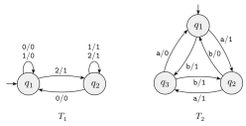
However, there is a much more elegant way of solving this problem. Suppose L were regular. Then, by Theorem 4.1, L and the language

$$L_1 = L \cap L\left(a^* b^*\right)$$

would also be regular. But $L_1 = \{a^n b^n : n \geq 0\}$, which we have already classified as nonregular. Consequently, L cannot be regular.

Finite State Transducer (FST) [S-ToC-13]

A ***finite state transducer*** (FST) is a type of deterministic finite automaton whose output is a string and not just *accept* or *reject*. The following are state diagrams of finite state transducers T_1 and T_2.



Each transition of an FST is labeled with two symbols, one designating the input symbol for that transition and the other designating the output symbol. The two symbols are written with a slash, /, separating them. In T_1, the transition from q_1 to q_2 has input symbol 2 and output symbol 1. Some transitions may have multiple input–output pairs, such as the transition in T_1 from q_1 to itself. When an FST computes on an input string w, it takes the input symbols $w_1 \cdots w_n$ one by one and, starting at the start state, follows the transitions by matching the input labels with the sequence of symbols $w_1 \cdots w_n = w$. Every time it goes along a transition, it outputs the corresponding output symbol. For example, on input 2212011, machine T_1 enters the sequence of states $q_1, q_2, q_2, q_2, q_2, q_1, q_1, q_1$ and produces output 1111000. On input abbb, T_2 outputs 1011. Give the sequence of states entered and the output produced in each of the following parts.

a. T_1 on input 011
b. T_1 on input 211
c. T_1 on input 121
d. T_1 on input 0202
e. T_2 on input b
f. T_2 on input bbab
g. T_2 on input bbbbbb
h. T_2 on input ε

An Introduction to the Theory of Computation

- Enumerability and Diagonalization
- Finite Automata and Regular Languages
- Pushdown Automata and Context-Free Languages
- Computation: Turing Machines
- Computation: Turing-Computability (Turing-Decidability)
- Computation: Reducibility (Turing-Reducibility)
- Computation: Recursive Functions
- Computation: Recursive Sets and Relations
- Equivalent Definitions of Computability
- Advanced Topics in Computability Theory
- Computational Complexity
- Time Complexity
- Space Complexity
- Intractability
- Advanced Topics in Complexity Theory

Grammars: Informal Definition

文法包括了一系列替代规则

♣ Grammar 文法

- [D] A *grammar* consists of a collection of *substitution rules*, also called *productions*.
- [D] Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a *variable*. The string consists of variables and other symbols called *terminals*.
- The variable symbols often are represented by capital letters. The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols.
- [D] One variable is designated as the *start variable*. It usually occurs on the left-hand side of the topmost rule.

Using a Grammar to Describe a Language

♣ Using a grammar to describe a language

- By generating each string of that language in the following manner:
  (1) Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.
  (2) Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
  (3) Repeat step 2 until no variables remain.

♣ The language of a grammar

- [D] All strings generated in this way constitute the *language of the grammar*.
- [D] The sequence of substitutions to obtain a string is called a *derivation*. The derivation can also be represented by a *derivation (parse) tree*.

Context-Free Grammars/Languages: Example G_1

The following is an example of a context-free grammar, which we call G_1.

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

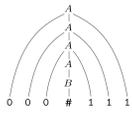


FIGURE 2.1
Parse tree for 000#111 in grammar G_1

- Grammar G_1 contains three rules.
- G_1's variables are A and B, where A is the start variable. Its terminals are 0, 1, and # .
- For example, grammar G_1 generates the string 000#111.
- A derivation of string 000#111 in grammar G_1 is
  A ⇒ 0A1 ⇒ 00A11 ⇒ 000A111 ⇒ 000B111 ⇒ 000#111.

Context-Free Grammars/Languages: Example G_2

The following is a second example of a context-free grammar, called G_2, which describes a fragment of the English language.

⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⟨NOUN-PHRASE⟩ → ⟨CMPLX-NOUN⟩ | ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩
⟨VERB-PHRASE⟩ → ⟨CMPLX-VERB⟩ | ⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩
⟨PREP-PHRASE⟩ → ⟨PREP⟩⟨CMPLX-NOUN⟩
⟨CMPLX-NOUN⟩ → ⟨ARTICLE⟩⟨NOUN⟩
⟨CMPLX-VERB⟩ → ⟨VERB⟩ | ⟨VERB⟩⟨NOUN-PHRASE⟩
⟨ARTICLE⟩ → a | the
⟨NOUN⟩ → boy | girl | flower
⟨VERB⟩ → touches | likes | sees
⟨PREP⟩ → with

⟨SENTENCE⟩ ⇒ ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⇒ ⟨CMPLX-NOUN⟩⟨VERB-PHRASE⟩
⇒ ⟨ARTICLE⟩⟨NOUN⟩⟨VERB-PHRASE⟩
⇒ a ⟨NOUN⟩⟨VERB-PHRASE⟩
⇒ a boy ⟨VERB-PHRASE⟩
⇒ a boy ⟨CMPLX-VERB⟩
⇒ a boy ⟨VERB⟩
⇒ a boy sees

- Grammar G_2 describes a fragment of the English language; it has 10 variables (the capitalized grammatical terms written inside brackets); 27 terminals (the standard English alphabet plus a space character); and 18 rules.

Context-Free Grammars (CFGs): Formal Definition [S-ToC-13]

DEFINITION 2.2

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

If u, v, and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv ***yields*** uwv, written $uAv \Rightarrow uwv$. Say that u ***derives*** v, written $u \overset{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

The ***language of the grammar*** is $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.

Context-Free Grammars (CFGs): Formal Definition [S-ToC-13]

♣ Notes

- Often we specify a grammar by writing down only its rules.
- We can identify the variables as the symbols that appear on the left-hand side of the rules and the terminals as the remaining symbols.
- By convention, the start variable is the variable on the left-hand side of the first rule.
- For convenience, when presenting a context-free grammar, we abbreviate several rules with the same left-hand variable, such as A → 0A1 and A → B, into a single line A → 0A1 | B, using the symbol "|" as an "or".

Context-Free Grammars (CFGs): Formal Definition [L-ToC-17]

DEFINITION 5.1

A grammar $G = (V, T, S, P)$ is said to be **context-free** if all productions in P have the form

$$A \to x,$$

where $A \in V$ and $x \in (V \cup T)^*$.

A language L is said to be context-free if and only if there is a context-free grammar G such that $L = L(G)$.

DEFINITION 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a **simple grammar** or **s-grammar** if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Context-Free Grammars (CFGs): Formal Definition [L-ToC-17]

DEFINITION 5.3

Let $G = (V, T, S, P)$ be a context-free grammar. An ordered tree is a derivation tree for G if and only if it has the following properties.

1. The root is labeled S.
2. Every leaf has a label from $T \cup \{\lambda\}$.
3. Every interior vertex (a vertex that is not a leaf) has a label from V.
4. If a vertex has label $A \in V$, and its children are labeled (from left to right) $a_1, a_2, ..., a_n$, then P must contain a production of the form
$$A \to a_1 a_2 \cdots a_n.$$
5. A leaf labeled λ has no siblings, that is, a vertex with a child labeled λ can have no other children.

A tree that has properties 3, 4, and 5, but in which 1 does not necessarily hold and in which property 2 is replaced by

2a. Every leaf has a label from $V \cup T \cup \{\lambda\}$,

is said to be a **partial derivation tree**.

The string of symbols obtained by reading the leaves of the tree from left to right, omitting any λ's encountered, is said to be the **yield** of the tree. The descriptive term *left to right* can be given a precise meaning. The yield is the string of terminals in the order they are encountered when the tree is traversed in a depth-first manner, always taking the leftmost unexplored branch.

Context-Free Grammars (CFGs): Derivation Theorems [L-ToC-17]

THEOREM 5.1

Let $G = (V, T, S, P)$ be a context-free grammar. Then for every $w \in L(G)$, there exists a derivation tree of G whose yield is w. Conversely, the yield of any derivation tree is in $L(G)$. Also, if t_G is any partial derivation tree for G whose root is labeled S, then the yield of t_G is a sentential form of G.

THEOREM 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B,$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

THEOREM 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

Context-Free Grammars/Languages: Example G_3 [S-ToC-13]

Consider grammar $G_3 = (\{S\}, \{\mathtt{a}, \mathtt{b}\}, R, S)$. The set of rules, R, is

$$S \to \mathtt{a}S\mathtt{b} \mid SS \mid \varepsilon.$$

This grammar generates strings such as abab, aaabbb, and aababb. You can see more easily what this language is if you think of a as a left parenthesis "(" and b as a right parenthesis ")". Viewed in this way, $L(G_3)$ is the language of all strings of properly nested parentheses. Observe that the right-hand side of a rule may be the empty string ε.

Context-Free Grammars/Languages: Example G_4 [S-ToC-13]

Consider grammar $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.
V is $\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ and Σ is $\{\mathtt{a}, \mathtt{+}, \mathtt{x}, \mathtt{(}, \mathtt{)}\}$. The rules are

$$\langle \text{EXPR} \rangle \to \langle \text{EXPR} \rangle \mathtt{+} \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle$$
$$\langle \text{TERM} \rangle \to \langle \text{TERM} \rangle \mathtt{x} \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle$$
$$\langle \text{FACTOR} \rangle \to \mathtt{(} \langle \text{EXPR} \rangle \mathtt{)} \mid \mathtt{a}$$

The two strings a+axa and (a+a)xa can be generated with grammar G_4. The parse trees are shown in the following figure.

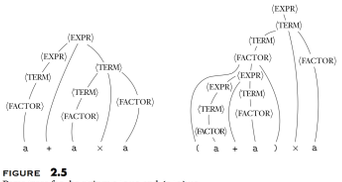


FIGURE **2.5**
Parse trees for the strings a+axa and (a+a)xa

定义规则同时
也定义了优先级

Context-Free Grammars/Languages: Examples [L-ToC-17]

EXAMPLE 5.1

The grammar $G = (\{S\}, \{a, b\}, S, P)$, with productions

$$S \to aSa,$$
$$S \to bSb,$$
$$S \to \lambda,$$

is context-free. A typical derivation in this grammar is

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbaa.$$

This, and similar derivations, make it clear that

$$L(G) = \left\{ww^R : w \in \{a, b\}^*\right\}.$$

The language is context-free, but as shown in Example 4.8, it is not regular.

Context-Free Grammars/Languages: Examples [L-ToC-17]

EXAMPLE 5.2

The grammar G, with productions

$$S \to abB,$$
$$A \to aaBb,$$
$$B \to bbAa,$$
$$A \to \lambda,$$

is context-free. We leave it to the reader to show that

$$L(G) = \{ab(bbaa)^n bba(ba)^n : n \geq 0\}.$$

Context-Free Grammars/Languages: Examples [L-ToC-17]

EXAMPLE 5.3

The language

$$L = \{a^n b^m : n \neq m\}$$

is context-free.

To show this, we need to produce a context-free grammar for the language. The case of $n = m$ is solved in Example 1.11 and we can build on that solution. Take the case $n > m$. We first generate a string with an equal number of a's and b's, then add extra a's on the left. This is done with

$$S \to AS_1,$$
$$S_1 \to aS_1b|\lambda,$$
$$A \to aA|a.$$

We can use similar reasoning for the case $n < m$, and we get the answer

$$S \to AS_1|S_1B,$$
$$S_1 \to aS_1b|\lambda,$$
$$A \to aA|a,$$
$$B \to bB|b.$$

The resulting grammar is context-free, hence L is a context-free language. However, the grammar is not linear.

Context-Free Grammars/Languages: Examples [L-ToC-17]

EXAMPLE 5.4

Consider the grammar with productions

$$S \to aSb|SS|\lambda.$$

This is another grammar that is context-free, but not linear. Some strings in $L(G)$ are $abaabb$, $aababb$, and $ababab$. It is not difficult to conjecture and prove that

$$L = \{w \in \{a,b\}^* : n_a(w) = n_b(w) \text{ and } n_a(v) \geq n_b(v), \text{ where } v \text{ is any prefix of } w\}. \quad (5.1)$$

We can see the connection with programming languages clearly if we replace a and b with left and right parentheses, respectively. The language L includes such strings as $(())$ and $()()()$ and is in fact the set of all properly nested parenthesis structures for the common programming languages.

Here again there are many other equivalent grammars. But, in contrast to Example 5.3, it is not so easy to see if there are any linear ones. We will have to wait until Chapter 8 before we can answer this question.

Context-Free Grammars/Languages: Examples [L-ToC-17]

EXAMPLE 5.6

Consider the grammar G, with productions

$$S \to aAB,$$
$$A \to bBb,$$
$$B \to A|\lambda.$$

The tree in Figure 5.2 is a partial derivation tree for G, while the tree in Figure 5.3 is a derivation tree. The string $aAbBB$, which is the yield of the first tree, is a sentential form of G. The yield of the second tree, $abbbb$, is a sentence of $L(G)$.

FIGURE 5.2

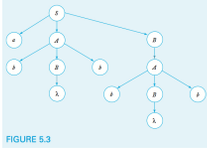


FIGURE 5.3

Context-Free Grammars/Languages: Examples [L-ToC-17]

EXAMPLE 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

Designing CFGs: Techniques

- **Many CFLs are the union of simpler CFLs**
  - **If you must construct a CFG for a CFL that you can break into simpler pieces, do so and then construct individual grammars for each piece.**
  - **These individual grammars can be easily merged into a grammar for the original language by combining their rules and then adding the new rule $S \to S_1 \mid S_2 \mid \cdots \mid S_k$, where the variables S_i are the start variables for the individual grammars.**

Designing CFGs: Techniques

♣ **Construct a DFA for the language at first**

- Constructing a CFG for a language that is regular is easy if you can first construct a DFA for that language.
- Any DFA can be converted into an equivalent CFG as follows:
- (1) Make a variable R_i for each state q_i of the DFA.
- (2) Add the rule $R_i \rightarrow aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA.
- (3) Add the rule $R_i \rightarrow \varepsilon$ if q_i is an accept state of the DFA.
- (4) Make R_0 the start variable of the grammar, where q_0 is the start state of the machine.

Designing CFGs: Techniques

♣ **Some CFLs contain strings with two "linked" substrings**

- Certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring.
- This situation occurs in the language $\{0^n1^n \mid n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s.
- We can construct a CFG to handle this situation by using a rule of the form $R \rightarrow uRv$, which generates strings wherein the portion containing the u's corresponds to the portion containing the v's.

Designing CFGs: Techniques

♣ **Recursive structures**

- In some more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures.
- That situation occurs in the grammar that generates arithmetic expressions in example G_4.
- Any time the symbol a appears, an entire parenthesized expression might appear recursively instead.
- To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.

Context-Free Grammars (CFGs): Ambiguity

♣ **Ambiguity** 二义性

- [D] If a grammar generates the same string in several different ways, we say that the string is derived *ambiguously* in that grammar.
- [D] If a grammar generates some string ambiguously, we say that the grammar is *ambiguous*.

若同一字符串能通过多种不同的方式生成，则该字符串是二义生成的

♣ **Inherently ambiguous languages**

- Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language.
- [D] Some context-free languages, however, can be generated only by ambiguous grammars. Such languages are called *inherently ambiguous*. (Ex: $\{a^ib^jc^k \mid i=j \text{ or } j=k\}$)

固有二义的

Context-Free Grammars (CFGs): Ambiguity Examples

没有优先级会产生二义

♣ **An example**

- Grammar G_5 generates the string a + a × a ambiguously.

For example, consider grammar G_5:

⟨EXPR⟩ → ⟨EXPR⟩+⟨EXPR⟩ | ⟨EXPR⟩×⟨EXPR⟩ | (⟨EXPR⟩) | a

This grammar generates the string a+a×a ambiguously. The following figure shows the two different parse trees.

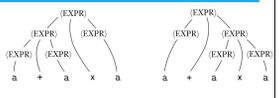


FIGURE 2.6
The two parse trees for the string a+a×a in grammar G_5

- Grammar G_5 does not capture the usual precedence relations and so may group the + before the × or vice versa.
- In contrast, grammar G_4 generates exactly the same language, but every generated string has a unique parse tree. Hence G_4 is unambiguous, whereas G_5 is ambiguous.

Context-Free Grammars (CFGs): Ambiguity Examples

♣ **Other examples**

- Grammar G_2 is another example of an ambiguous grammar. The sentence "the girl touches the boy with the flower" has two different derivations.
- The language $\{ a^ib^jc^k \mid i = j \text{ or } j = k \}$ is inherently ambiguous.

Context-Free Grammars (CFGs): Ambiguity Definition

♣ **Definition of ambiguity**

DEFINITION 2.7

A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

展开变量时，总是展开最左边的

♣ **Leftmost derivation** 最左展开

◆ **[D] A derivation of a string w in a grammar G is a *leftmost derivation* if at every step the leftmost remaining variable is the one replaced.**

DEFINITION 5.2

A derivation is said to be **leftmost** if in each step the leftmost variable in the sentential form is replaced. If in each step the rightmost variable is replaced, we call the derivation **rightmost**.

Context-Free Grammars (CFGs): Ambiguity Definitions

DEFINITION 5.5

A context-free grammar G is said to be **ambiguous** if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

DEFINITION 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be unambiguous. If every grammar that generates L is ambiguous, then the language is called **inherently ambiguous**.

CFGs: Leftmost and Rightmost Derivations [L-ToC-17]

EXAMPLE 5.5

Consider the grammar with productions

$$S \to aAB,$$
$$A \to bBb,$$
$$B \to A|\lambda.$$

Then

$$S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abAbB \Rightarrow abbBbbB \Rightarrow abbbbB \Rightarrow abbbb$$

is a leftmost derivation of the string $abbbb$. A rightmost derivation of the same string is

$$S \Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abAb \Rightarrow abbBbb \Rightarrow abbbb.$$

Context-Free Grammars (CFGs): Ambiguity Examples [L-ToC-17]

EXAMPLE 5.10

The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous. The sentence $aabb$ has the two derivation trees shown in Figure 5.4.

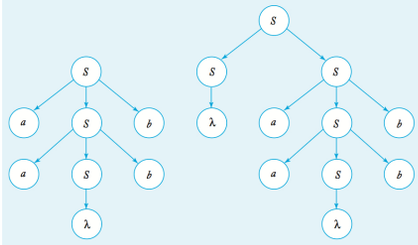


FIGURE 5.4

Context-Free Grammars (CFGs): Ambiguity Examples [L-ToC-17]

EXAMPLE 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\},$$
$$T = \{a, b, c, +, *, (,)\},$$

and productions

$$E \to I,$$
$$E \to E + E,$$
$$E \to E * E,$$
$$E \to (E),$$
$$I \to a|b|c.$$

The strings $(a + b)*c$ and $a*b + c$ are in $L(G)$. It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C-like programming languages. The grammar is ambiguous. For instance, the string $a + b*c$ has two different derivation trees, as shown in Figure 5.5.

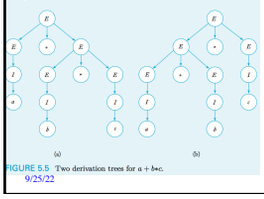


FIGURE 5.5 Two derivation trees for $a + b*c$.

EXAMPLE 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in Figure 5.6. No other derivation tree is possible for this string. The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we will later show that there are no general algorithms by which these questions can always be resolved.

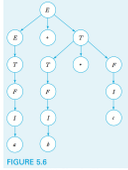


FIGURE 5.6

Context-Free Grammars (CFGs): Ambiguity Examples [L-ToC-17]

EXAMPLE 5.13

The language

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\},$$

with n and m nonnegative, is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where L_1 is generated by

$$S_1 \to S_1 c|A,$$
$$A \to aAb|\lambda$$

and L_2 is given by an analogous grammar with start symbol S_2 and productions

$$S_2 \to aS_2|B,$$
$$B \to bBc|\lambda.$$

Then L is generated by the combination of these two grammars with the additional production

$$S \to S_1|S_2.$$

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison 1978.

Simplification of CFGs [L-ToC-17]

♣A useful substitution rule

THEOREM 6.1

Let $G = (V, T, S, P)$ be a context-free grammar. Suppose that P contains a production of the form

$$A \to x_1 B x_2.$$

Assume that A and B are different variables and that

$$B \to y_1 | y_2 | \cdots | y_n$$

is the set of all productions in P that have B as the left side. Let $\widehat{G} = \left(V, T, S, \widehat{P}\right)$ be the grammar in which $\widehat{P}$ is constructed by deleting

$$A \to x_1 B x_2 \tag{6.1}$$

from P, and adding to it

$$A \to x_1 y_1 x_2 | x_1 y_2 x_2 | \cdots | x_1 y_n x_2.$$

Then

$$L\left(\widehat{G}\right) = L(G).$$

- **Theorem 6.1 is a simple and quite intuitive substitution rule: A production A → x1Bx2 can be eliminated from a grammar if we put in its place the set of productions in which B is replaced by all strings it derives in one step. In this result, it is necessary that A and B be different variables**

Simplification of CFGs [L-ToC-17]

♣Removing useless productions

DEFINITION 6.1

Let $G = (V, T, S, P)$ be a context-free grammar. A variable $A \in V$ is said to be **useful** if and only if there is at least one $w \in L(G)$ such that

$$S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} w, \tag{6.2}$$

with x, y in $(V \cup T)^*$. In words, a variable is useful if and only if it occurs in at least one derivation. A variable that is not useful is called **useless**. A production is useless if it involves any useless variable.

THEOREM 6.2

Let $G = (V, T, S, P)$ be a context-free grammar. Then there exists an equivalent grammar $\widehat{G} = \left(\widehat{V}, \widehat{T}, S, \widehat{P}\right)$ that does not contain any useless variables or productions.

Simplification of CFGs [L-ToC-17]

♣Removing λ-productions

DEFINITION 6.2

Any production of a context-free grammar of the form

$$A \to \lambda$$

is called a **λ-production**. Any variable A for which the derivation

$$A \overset{*}{\Rightarrow} \lambda \tag{6.3}$$

is possible is called **nullable**.

THEOREM 6.3

Let G be any context-free grammar with λ not in $L(G)$. Then there exists an equivalent grammar $\widehat{G}$ having no λ-productions.

Simplification of CFGs [L-ToC-17]

♣Removing unit-productions

DEFINITION 6.3

Any production of a context-free grammar of the form

$$A \to B,$$

where $A, B \in V$, is called a **unit-production**.

THEOREM 6.4

Let $G = (V, T, S, P)$ be any context-free grammar without λ-productions. Then there exists a context-free grammar $\widehat{G} = \left(\widehat{V}, \widehat{T}, S, \widehat{P}\right)$ that does not have any unit-productions and that is equivalent to G.

♣Removing useless productions, λ-productions, unit-productions

THEOREM 6.5

Let L be a context-free language that does not contain λ. Then there exists a context-free grammar that generates L and that does not have any useless productions, λ-productions, or unit-productions.

Context-Free Grammars (CFGs): Chomsky Normal Form [S-ToC-13]

♣Chomsky normal form

Chomsky 范式

DEFINITION 2.8

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \to BC$$
$$A \to a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \to \varepsilon$, where S is the start variable.

♣Theorem (Chomsky normal form)

THEOREM 2.9

Any context-free language is generated by a context-free grammar in Chomsky normal form.

任意上下文无关语言能由

Chomsky范式的上下文无关文法生成

Context-Free Grammars (CFGs): Chomsky Normal Form [L-ToC-17]

♣Chomsky normal form

DEFINITION 6.4

A context-free grammar is in Chomsky normal form if all productions are of the form

$$A \to BC$$

or

$$A \to a,$$

where A, B, C are in V, and a is in T.

♣Theorem (Chomsky normal form)

THEOREM 6.6

Any context-free grammar $G = (V, T, S, P)$ with $\lambda \notin L(G)$ has an equivalent grammar $\widehat{G} = \left(\widehat{V}, \widehat{T}, S, \widehat{P}\right)$ in Chomsky normal form.

Context-Free Grammars (CFGs): Chomsky Normal Form [L-ToC-17]

♣ Greibach normal form

DEFINITION 6.5

A context-free grammar is said to be in Greibach normal form if all productions have the form

$$A \to ax,$$

where $a \in T$ and $x \in V^*$.

♣ Theorem (Greibach normal form)

THEOREM 6.7

For every context-free grammar G with $\lambda \notin L(G)$, there exists an equivalent grammar $\hat{G}$ in Greibach normal form.

(Nondeterministic) Pushdown Automata (PDAs)

♣ (Nondeterministic) Pushdown automata (PDAs) 非确定性下推自动机

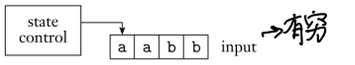
- [D] *(nondeterministic) Pushdown automata (PDAs)* are like nondeterministic finite automata but have an extra component called a *stack*. The stack provides additional unlimited (infinite) memory beyond the finite amount available in the control.
- The stack allows pushdown automata to recognize some non-regular languages.

♣ Fact

- (Nondeterministic) PDAs are equivalent in power to CFGs. NPDA与CFG等价
- This equivalence is useful because it gives us two options for proving that a language is context free. We can give either a CFG generating it or a PDA recognizing it.

(Nondeterministic) PDAs: An Example [S-ToC-13]

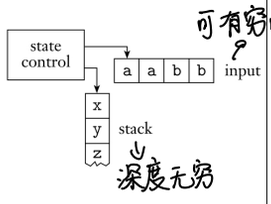
有穷状态自动机
不能处理无穷输入



→有穷

FIGURE 2.11
Schematic of a finite automaton

可有穷可无穷?



深度无穷

FIGURE 2.12
Schematic of a pushdown automaton

(Nondeterministic) Pushdown Automata (PDAs)

♣ Pushing and popping

- A PDA can *write* symbols on the stack and read them back later.
- Writing a symbol "*pushes down*" all the other symbols on the stack.
- At any time the symbol on the top of the stack can be read and removed. The remaining symbols then move back up.
- *Writing* a symbol on the stack is often referred to as *pushing* the symbol, and *removing* a symbol is referred to as *popping* it.
- Note: All access to the stack, for both reading and writing, may be done only at the top. In other words a stack is a "*last in, first out*" storage device.

(Nondeterministic) Pushdown Automata (PDAs)

♣ The role of the stack

- A stack is valuable because it can hold an unlimited amount of data. The unlimited nature of a stack allows the PDA to store numbers of unbounded size.
- A finite automaton is unable to recognize the language $\{0^n1^n \mid n \geq 0\}$ because it cannot store very large numbers in its finite memory.
- A PDA is able to recognize this language because it can use its stack to store the number of 0s it has seen.

(Nondeterministic) Pushdown Automata (PDAs): An Example

♣ Example of recognizing $\{0^n1^n \mid n \geq 0\}$

- Read symbols from the input.
- As each 0 is read, push it onto the stack.
- As soon as 1s are seen, pop a 0 off the stack for each 1 read.
- If reading the input is finished exactly when the stack becomes empty of 0s, accept the input.
- If the stack becomes empty while 1s remain or if the 1s are finished while the stack still contains 0s or if any 0s appear in the input following 1s, reject the input.

当输入结束时，栈空了，则接受
当栈空时仍有1或输入结束时栈未空，则拒绝

(Nondeterministic) Pushdown Automata (PDAs): Formal Definition [S-ToC-13]

DEFINITION 2.13

A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

在状态转移函数中允许有空串

♣ **Transition function δ: $(Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon) \rightarrow P(Q \times \Gamma_\varepsilon)$ (Power set!)**

◆ **$(q_i, a, b) \rightarrow (q_{i+1}, c)$ means that for a state q_i, the current input symbol a, and the top stack symbol b, the next state and top stack symbol will be q_{i+1} and c.**

a,b→c 表示从栈中弹出b，写入c

a,ε→c 表示不从栈中弹出，直接写入c

a b→ε 表示从栈中弹出b，不写入

(Nondeterministic) Pushdown Automata (PDAs): Formal Definition [S-ToC-13]

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. It accepts input w if w can be written as $w = w_1 w_2 \cdots w_m$, where each $w_i \in \Sigma_\varepsilon$ and sequences of states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ exist that satisfy the following three conditions. The strings s_i represent the sequence of stack contents that M has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = \varepsilon$. This condition signifies that M starts out properly, in the start state and with an empty stack.
2. For $i = 0, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol.
3. $r_m \in F$. This condition states that an accept state occurs at the input end.

♣ **Transition function δ: $(Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon) \rightarrow P(Q \times \Gamma_\varepsilon)$ (Power set!)**

◆ **$(q_i, a, b) \rightarrow (q_{i+1}, c)$ means that for a state q_i, the current input symbol a, and the top stack symbol b, the next state and top stack symbol will be q_{i+1} and c.**

Nondeterministic Pushdown Accepters (NPDAs): Formal Definition [L-ToC-17]

DEFINITION 7.1

A **nondeterministic pushdown accepter (npda)** is defined by the septuple

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

where

Q is a finite set of internal states of the control unit,

Σ is the input alphabet,

Γ is a finite set of symbols called the **stack alphabet**,

$\delta: Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow$ set of finite subsets of $Q \times \Gamma^*$ is the transition function,

$q_0 \in Q$ is the initial state of the control unit,

$z \in \Gamma$ is the **stack start symbol**,

$F \subseteq Q$ is the set of final states.

(Nondeterministic) Pushdown Automata (PDAs): Example M_1 [S-ToC-13]

EXAMPLE 2.14

The following is the formal description of the PDA (page 112) that recognizes the language $\{0^n 1^n \mid n \geq 0\}$. Let M_1 be $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where

$Q = \{q_1, q_2, q_3, q_4\}$,

$\Sigma = \{0, 1\}$,

$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0			1			ε		
Stack:	0	$	ε	0	$	ε	0	$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									

有q_1是为了确保输入是空串，直接接受的情况

若输入不为空串，则先压一个$入栈，进入$q_2$，并开始读输入串.

(Nondeterministic) Pushdown Automata (PDAs): Example M_1 [S-ToC-13]

to state. We write "$a,b \rightarrow c$" to signify that when the machine is reading an a from the input, it may replace the symbol b on the top of the stack with a c. Any of a, b, and c may be ε. If a is ε, the machine may make this transition without reading any symbol from the input. If b is ε, the machine may make this transition without reading and popping any symbol from the stack. If c is ε, the machine does not write any symbol on the stack when going along this transition.

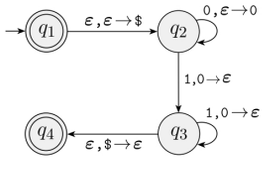


FIGURE 2.15
State diagram for the PDA M_1 that recognizes $\{0^n 1^n \mid n \geq 0\}$

(Nondeterministic) Pushdown Automata (PDAs): Example M_1 [S-ToC-13]

The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack. This PDA is able to get the same effect by initially placing a special symbol $ on the stack. Then if it ever sees the $ again, it knows that the stack effectively is empty. Subsequently, when we refer to testing for an empty stack in an informal description of a PDA, we implement the procedure in the same way.

Similarly, PDAs cannot test explicitly for having reached the end of the input string. This PDA is able to achieve that effect because the accept state takes effect only when the machine is at the end of the input. Thus from now on, we assume that PDAs can test for the end of the input, and we know that we can implement it in the same manner.

(Nondeterministic) Pushdown Automata (PDAs): Example M_2 [S-ToC-13]

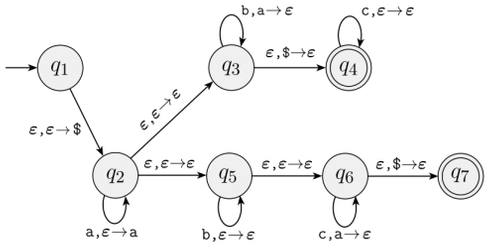


FIGURE 2.17
State diagram for PDA M_2 that recognizes $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

(Nondeterministic) Pushdown Automata (PDAs): Example M_3 [S-ToC-13]

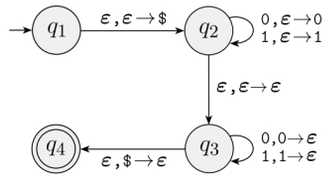


FIGURE 2.19
State diagram for the PDA M_3 that recognizes $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$

上下文无关语言
⇔有NPDA识别

Equivalence between CFGs and (Nondeterminstic) PDAs [S-ToC-13]

THEOREM 2.20

A language is context free if and only if some pushdown automaton recognizes it.

LEMMA 2.21

If a language is context free, then some pushdown automaton recognizes it.

LEMMA 2.27

If a pushdown automaton recognizes some language, then it is context free.

CLAIM 2.30

If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack.

CLAIM 2.31

If x can bring P from p with empty stack to q with empty stack, A_{pq} generates x.

Equivalence between CFLs and NPDAs [L-ToC-17]

DEFINITION 7.2

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ be a nondeterministic pushdown automaton. The language accepted by M is the set

$$L(M) = \left\{ w \in \Sigma^* : (q_0, w, z) \overset{*}{\vdash}_M (p, \lambda, u), p \in F, u \in \Gamma^* \right\}.$$

In words, the language accepted by M is the set of all strings that can put M into a final state at the end of the string. The final stack content u is irrelevant to this definition of acceptance.

THEOREM 7.1

For any context-free language L, there exists an npda M such that

$$L = L(M).$$

THEOREM 7.2

If $L = L(M)$ for some npda M, then L is a context-free language.

Relationship between RLs and CFLs [S-ToC-13]

COROLLARY 2.32

Every regular language is context free.

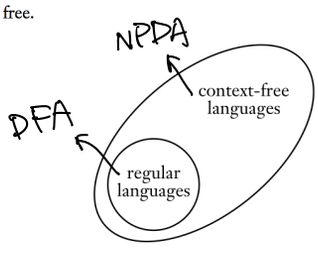


FIGURE 2.33
Relationship of the regular and context-free languages

The Algorithmic Problems Related to Context-Free Languages [LP-ToC-98]

Theorem 3.6.1: *(a) There is a polynomial algorithm which, given a context-free grammar, constructs an equivalent pushdown automaton.*
(b) There is a polynomial algorithm which, given a pushdown automaton, constructs an equivalent context-free grammar.
(c) There is a polynomial algorithm which, given a context-free grammar G and a string x, decides whether $x \in L(G)$.

上下文无关语言的 pumping lemma

Non-Context-Free Languages

♣ **The Pumping Lemma for CFLs**

THEOREM **2.34**

Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

♣ **The idea to prove that a language is not context-free**

- **To use the pumping lemma to prove that a language B is not context-free, first assume that B is context-free and should satisfy the pumping lemma, then obtain a contradiction to show the assumption does not hold.**

Non-Context-Free Languages: Examples

- **The language B = { $a^nb^nc^n \mid n \geq 0$ } is not context free.**
- **The language C = { $a^ib^jc^k \mid 0 \leq i \leq j \leq k$ } is not context free.**
- **The language D = { $ww \mid w \in \{0, 1\}^*$ } is not context free.**

确定性下推自动机

Deterministic Pushdown Automata (DPDAs): Formal Definition [S-ToC-13]

DEFINITION **2.39**

A ***deterministic pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow (Q \times \Gamma_\varepsilon) \cup \{\emptyset\}$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

允许为空

The transition function δ must satisfy the following condition. For every $q \in Q$, $a \in \Sigma$, and $x \in \Gamma$, exactly one of the values

$$\delta(q, a, x), \delta(q, a, \varepsilon), \delta(q, \varepsilon, x), \text{ and } \delta(q, \varepsilon, \varepsilon)$$

is not $\emptyset$.

Deterministic Pushdown Automata (DPDAs): Formal Definition [L-ToC-17]

DEFINITION 7.3

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ is said to be deterministic if it is an automaton as defined in Definition 7.1, subject to the restrictions that, for every $q \in Q, a \in \Sigma \cup \{\lambda\}$ and $b \in \Gamma$,

1. $\delta(q, a, b)$ contains at most one element,
2. if $\delta(q, \lambda, b)$ is not empty, then $\delta(q, c, b)$ must be empty for every $c \in \Sigma$.

The first of these conditions simply requires that for any given input symbol and any stack top, at most one move can be made. The second condition is that when a λ-move is possible for some configuration, no input-consuming alternative is available.

DEFINITION 7.4

A language L is said to be a **deterministic context-free language** if and only if there exists a dpda M such that $L = L(M)$.

Deterministic Pushdown Automata (DPDAs)

♣ **ε-moves**

- **ε-moves take two forms: *ε-input moves* corresponding to δ(*q*, *ε*, *x*), and ε-stack moves corresponding to δ(*q*, *a*, *ε*).**
- **A move may combine both forms, corresponding to δ(*q*, *ε*, *ε*).**

♣ **DPDAs vs DFAs**

- **Defining DPDAs is more complicated than defining DFAs because DPDAs may read an input symbol without popping a stack symbol, and vice versa.**
- **We allow ε-moves in the DPDA's transition function even though ε-moves are prohibited in DFAs.**

Deterministic Pushdown Automata (DPDAs)

♣ **The transition function**

- **The transition function may output either a single move of the form (*r*, *y*) or it may indicate no action by outputting ϕ.**
- **The condition of the transition function enforces deterministic behavior by preventing the DPDA from taking two different actions in the same situation.**
- **A DPDA has exactly one legal move in every situation where its stack is nonempty.**
- **If the stack is empty, a DPDA can move only if the transition function specifies a move that pops ε. Otherwise the DPDA has no legal move and it rejects without reading the rest of the input.**

Deterministic Context-Free Languages (DCFLs)

♣ **Acceptance for DPDAs**

- **If a DPDA enters an accept state after it has read the last input symbol of an input string, it accepts that string. In all other cases, it rejects that string.**
- **Rejection occurs if the DPDA reads the entire input but does not enter an accept state when it is at the end, or if the DPDA fails to read the entire input string. The latter case may arise if the DPDA tries to pop an empty stack or if the DPDA makes an endless sequence of ε-input moves without reading the input past a certain point.**

♣ **Deterministic context-free languages (DCFLs)**

- **[D] The languages that are recognizable by DPDAs are called *deterministic context-free languages (DCFLs)* that is a subclass of the context-free languages.**

Properties of DPDAs and DCFLs

♣ **DPDA that always reads the entire input string**

LEMMA 2.41

Every DPDA has an equivalent DPDA that always reads the entire input string.

♣ **Closure property of the class of DCFLs**

THEOREM 2.42

The class of DCFLs is closed under complementation.

♣ **Endmarked inputs and endmarked languages**

- **The special endmarker symbol ⊣ is appended to the input string. For any language A, we write the *endmarked language* $A⊣$ to be the collection of strings $w⊣$ where $w \in A$.**

THEOREM 2.43

A is a DCFL if and only if $A⊣$ is a DCFL.

Properties of CFLs [L-ToC-17]

THEOREM 8.1

Let L be an infinite context-free language. Then there exists some positive integer m such that any $w \in L$ with $|w| \geq m$ can be decomposed as

$$w = uvxyz, \tag{8.1}$$

with

$$|vxy| \leq m, \tag{8.2}$$

and

$$|vy| \geq 1, \tag{8.3}$$

such that

$$uv^ixy^iz \in L, \tag{8.4}$$

for all $i = 0, 1, 2, \ldots$. This is known as the pumping lemma for context-free languages.

Properties of CFLs [L-ToC-17]

DEFINITION 8.1

A context-free language L is said to be linear if there exists a linear context-free grammar G such that $L = L(G)$.

THEOREM 8.2

Let L be an infinite linear language. Then there exists some positive integer m, such that any $w \in L$, with $|w| \geq m$ can be decomposed as $w = uvxyz$ with

$$|uvyz| \leq m, \tag{8.5}$$
$$|vy| \geq 1, \tag{8.6}$$

such that

$$uv^ixy^iz \in L, \tag{8.7}$$

for all $i = 0, 1, 2, \ldots$.

Note that the conclusions of this theorem differ from those of Theorem 8.1, since (8.2) is replaced by (8.5). This implies that the strings v and y to be pumped must now be located within m symbols of the left and right ends of w, respectively. The middle string x can be of arbitrary length.

Properties of CFLs [L-ToC-17]

THEOREM 8.3

The family of context-free languages is closed under union, concatenation, and star-closure.

THEOREM 8.4

The family of context-free languages is not closed under intersection and complementation.

THEOREM 8.5

Let L_1 be a context-free language and L_2 be a regular language. Then $L_1 \cap L_2$ is context free.

THEOREM 8.6

Given a context-free grammar $G = (V, T, S, P)$, there exists an algorithm for deciding whether or not $L(G)$ is empty.

THEOREM 8.7

Given a context-free grammar $G = (V, T, S, P)$, there exists an algorithm for determining whether or not $L(G)$ is infinite.

Introduction to the Theory of Computation

- **Enumerability and Diagonalization**
- **Finite Automata and Regular Languages**
- **Pushdown Automata and Context-Free Languages**
- **Computation: Turing Machines**
- **Computation: Turing-Computability (Turing-Decidability)**
- **Computation: Reducibility (Turing-Reducibility)**
- **Computation: Recursive Functions**
- **Computation: Recursive Sets and Relations**
- **Equivalent Definitions of Computability**
- **Advanced Topics in Computability Theory**
- **Computational Complexity**
- **Time Complexity**
- **Space Complexity**
- **Intractability**
- **Advanced Topics in Complexity Theory**