# Modeling Computation
# แบบจำลองการทำงานของคอมพิวเตอร์

## Models of Computation

- Given a task:
  - Can it be carried out using a computer?
    - How can the task be carried out?
- Computation models can help answer these questions.
- Structures of computation models
  - Grammars
  - Finite-state machines
  - Turing machines

## Chapter Summary

1. Language and Grammars
2. Finite-State Machines with Output
3. Finite-State Machines with No Output
4. Language Recognition
5. Turing Machines

## 1. Language and Grammars

- Words in the English language can be combined in various ways.
- English **grammar** tells whether a combination of words is a valid sentence.
- For example, the sentence *the frog writes neatly* is a valid sentence according to the rules of English grammar.
    - *Swims quickly data*

**Syntax** (form of a sentence) vs **Semantics** (meaning of a sentence)

## Grammars

- The rules that specify the syntactically correct sentences of a ***natural language*** such as English, French, German, or Spanish, are extremely complex.
- Instead of studying natural languages, we can define ***formal languages*** that have well-defined rules of syntax.
- These rules of syntax are important both in linguistics, the study of natural languages, and in the study of programming languages.

## Grammars Example

A defined list of rules that describe how a valid sentence can be produced.

1. a **sentence** is made up of a **noun phrase** followed by a **verb phrase**;
2. a **noun phrase** is made up of an **article** followed by an **adjective** followed by a **noun**, or
3. a **noun phrase** is made up of an **article** followed by a **noun**;
4. a **verb phrase** is made up of a **verb** followed by an **adverb**, or
5. a **verb phrase** is made up of a **verb**;
6. an **article** is *a*, or
7. an **article** is *the*;
8. an **adjective** is *large*, or
9. an **adjective** is *hungry*;
10. a **noun** is *rabbit*, or
11. a **noun** is *student*;
12. a **verb** is *eats*, or
13. a **verb** is *hops*;
14. an **adverb** is *quickly*, or
15. an **adverb** is *wildly*.

## Phrase-Structure Grammars

**Terminology**

A grammar provides a set of **symbols** of various types and a set of rules for producing words.

- A ***vocabulary*** (or ***alphabet***) *V* is a finite, nonempty set of elements called ***symbols***.
- A ***word*** (or ***sentence***) over *V* is a string of finite length of elements of *V*.
- The ***empty string*** or ***null string***, denoted by $\lambda$ (and sometimes by $\epsilon$), is the string containing no symbols.
- The set of all words over *V* is denoted by *V*∗.
    - A ***language*** over V is a subset of V∗.

## Phrase-Structure Grammars

- The elements of *V* that can not be replaced by other symbols are called **terminals (T)**.
    - For instance, T = {*a*, *the*, *rabbit*, *student*, *hops*, *eats*, *quickly*, *wildly*}
- Those that can be replaced by other symbols are called **nonterminals (N)**.
    - For instance, N = {***sentence, noun phrase, verb phrase, adjective, article, noun, verb, adverb***}
- The ***start symbol***, denoted by ***S***, is the first element of vocabulary.

## Phrase-Structure Grammars

- The rules that specify when a string V* can be replaced with another string are called **productions** of the grammar.
  - The notation $z_0 \rightarrow z_1$ is the production that specifies that $z_0$ can be replaced by $z_1$ within a string.
  - The first production is **sentence → noun phrase verb phrase**.

## Phrase-Structure Grammars

- A ***phrase-structure grammar*** $G = (V, T, S, P)$ consists of a vocabulary $V$, a subset $T$ of $V$ consisting of terminal symbols, a start symbol $S$ from $V$, and a finite set of productions $P$.
- The set $N = V - T$ is the set of nonterminal symbols.
- Every production in $P$ must contain at least one nonterminal on its left side.

Example 1: Let $G = (V, T, S, P)$, where $V = \{a, b, A, B, S\}$, $T = \{a, b\}$, $S$ is the start symbol, and $P = \{S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, AB \rightarrow b\}$.

## Derivations

- Let $G = (V, T, S, P)$ be a phrase-structure grammar.
- Let $w_0 = lz_0r$ (that is, the concatenation of $l$, $z_0$, and $r$) and $w_1 = lz_1r$ be strings over $V$.
- If $z_0 \rightarrow z_1$ is a production of $G$, we say that $w_1$ is ***directly derivable*** from $w_0$ and we write $w_0 \Rightarrow w_1$.
- If $w_0, w_1, \ldots, w_n$ are strings over $V$ such that $w_0 \Rightarrow w_1, w_1 \Rightarrow w_2, \ldots, w_{n-1} \Rightarrow w_n$, then $w_n$ is ***derivable*** from $w_0$, and we write $w_0 \Rightarrow^* w_n$.
- The sequence of steps used to obtain $w_n$ from $w_0$ is called a **derivation**.

## Derivations Example

Example 2: Let $G = (V, T, S, P)$, where $V = \{a, b, A, B, S\}$, $T = \{a, b\}$, $S$ is the start symbol, and $P = \{S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, AB \rightarrow b\}$

- The string *Aaba* is directly derivable from *ABa* in the grammar.
  - because $B \rightarrow ab$ is a production in the grammar.

Is *abababa* a derivation from ABa?

## Language Generation

- Let $G = (V, T, S, P)$ be a phrase-structure grammar.
- The **language generated** by $G$ (or the **language** of $G$), denoted by $L(G)$, is the set of all strings of terminals that are derivable from the starting state $S$.
- In other words, $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$.

Example 3: Let $G$ be the grammar with vocabulary $V = \{S, A, a, b\}$, set of terminals $T = \{a, b\}$, starting symbol $S$, and productions $P = \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\}$. What is $L(G)$, the language of this grammar?

## Language Generation Exercise

Exercise 1: Let $G$ be the grammar with vocabulary $V = \{S, 0, 1\}$, set of terminals $T = \{0, 1\}$, starting symbol $S$, and productions $P = \{S \rightarrow 11S, S \rightarrow 0\}$.

What is L(G), the language of this grammar?

## Types of Phrase-Structure Grammars

©SASCHA SCHUERMANN/
AFP/Getty Images

- Phrase-structure grammars can be classified according to the types of productions that are allowed.
- This classification scheme, known as **Chomsky hierarchy,** is introduced by Noam Chomsky.

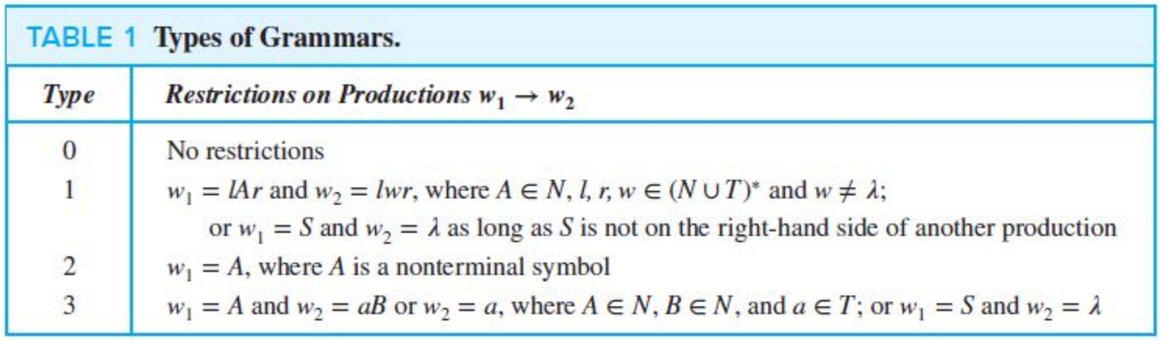
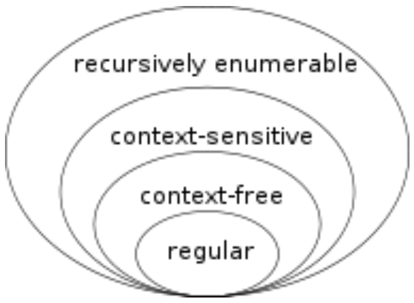
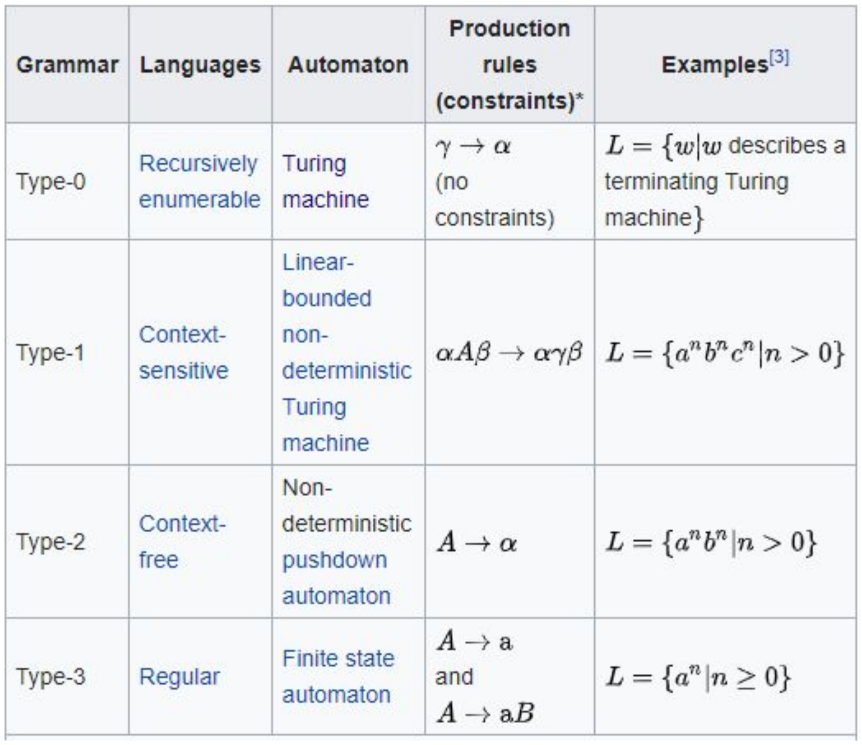


**TABLE 1 Types of Grammars.**

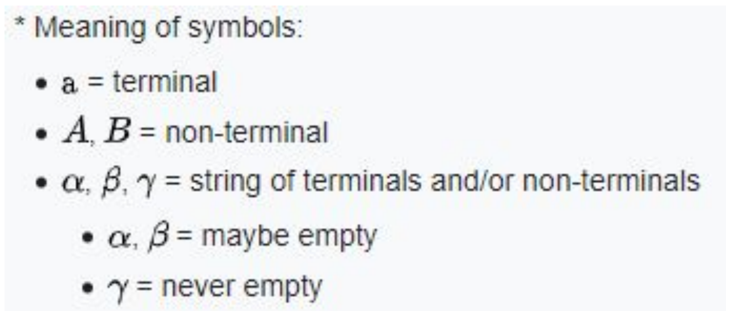
| *Type* | *Restrictions on Productions $w_1 \to w_2$* |
|---|---|
| 0 | No restrictions |
| 1 | $w_1 = lAr$ and $w_2 = lwr$, where $A \in N$, $l, r, w \in (N \cup T)^*$ and $w \neq \lambda$; or $w_1 = S$ and $w_2 = \lambda$ as long as $S$ is not on the right-hand side of another production |
| 2 | $w_1 = A$, where $A$ is a nonterminal symbol |
| 3 | $w_1 = A$ and $w_2 = aB$ or $w_2 = a$, where $A \in N$, $B \in N$, and $a \in T$; or $w_1 = S$ and $w_2 = \lambda$ |



## Types of Phrase-Structure Grammars



| Grammar | Languages | Automaton | Production rules (constraints)* | Examples[3] |
|---|---|---|---|---|
| Type-0 | Recursively enumerable | Turing machine | $\gamma \to \alpha$ (no constraints) | $L = \{w \mid w$ describes a terminating Turing machine$\}$ |
| Type-1 | Context-sensitive | Linear-bounded non-deterministic Turing machine | $\alpha A \beta \to \alpha \gamma \beta$ | $L = \{a^n b^n c^n \mid n > 0\}$ |
| Type-2 | Context-free | Non-deterministic pushdown automaton | $A \to \alpha$ | $L = \{a^n b^n \mid n > 0\}$ |
| Type-3 | Regular | Finite state automaton | $A \to \mathrm{a}$ and $A \to \mathrm{a}B$ | $L = \{a^n \mid n \geq 0\}$ |



* Meaning of symbols:

- a = terminal
- $A$, $B$ = non-terminal
- $\alpha$, $\beta$, $\gamma$ = string of terminals and/or non-terminals
  - $\alpha$, $\beta$ = maybe empty
  - $\gamma$ = never empty

## Types of Phrase-Structure Grammars

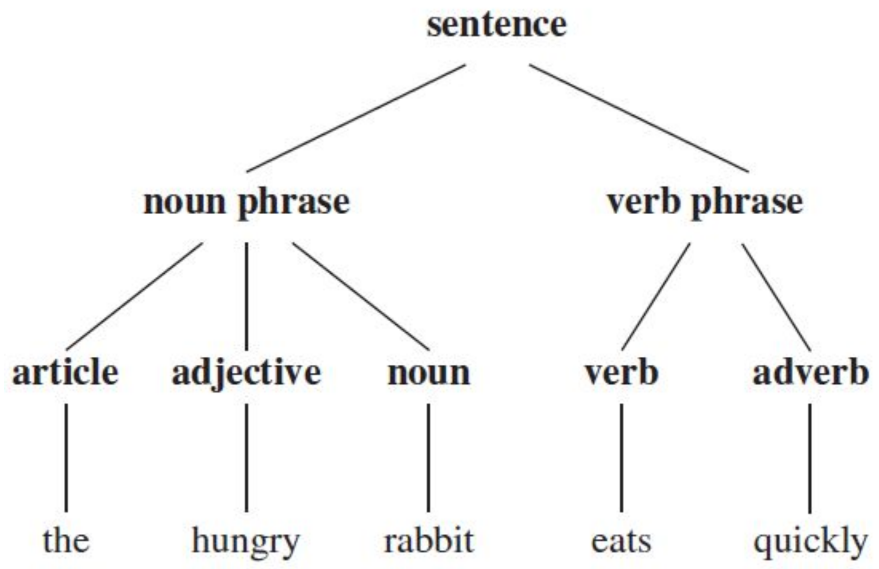
- Type 0 grammars include all formal grammars.
- Type 1 grammars generate **context-sensitive** languages.
- Type 2 grammars generate **context-free** languages.
  - Context-free languages are the theoretical basis for the phrase structure of most programming languages.
- Type-3 grammars generate the **regular** languages.
  - Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

## Derivation Trees

- A derivation in the language generated by a context-free grammar can be represented graphically using an ordered rooted tree, called a **derivation**, or **parse tree**.
  - The root of this tree represents the starting symbol.
  - The internal vertices of the tree represent the nonterminal symbols that arise in the derivation.
  - The leaves of the tree represent the terminal symbols that arise.
  - If the production $A \rightarrow w$ arises in the derivation, where $w$ is a word, the vertex that represents $A$ has as children vertices that represent each symbol in $w$, in order from left to right.

## Derivation Trees Example

- Example 4: Construct a derivation tree for the derivation of *the hungry rabbit eats quickly*?



## Derivation Trees Exercise

- Exercise 2: Construct a derivation tree for the derivation of *โดนัลทรัมป์สั่งแบนสินค้าจากประเทศไทย*

## What is Parsing?

- In computer science, **parsing** is the process of analysing text to determine if it belongs to a specific language or not (i.e. is syntactically valid for that language's grammar).
  - It is an informal name for the **syntactic analysis** process.
- The problem of determining whether a string is in the language generated by a context-free grammar arises in many applications, such as in the construction of compilers.

## What is Parsing?

- For example, suppose the language $a^n$ $b^n$ (which means same number of characters A followed by the same number of characters B).
  - A parser for that language would accept AABB input and reject the AAAB input.
- In addition, during this process a data structure could be created for further processing.
  - For instance, it could store the AA and BB in two separate stacks.
- This data structure can then be used by a compiler, interpreter or translator to create an executable program or library

## Parsing

- If you are given a sentence to break it down to parts of speech (nouns, verbs, etc.) and check whether it is grammatically correct, then you are parsing the sentence.
- When given a string, the naive approach for determining whether it is in the language generated by a grammar is to look for a sequence of productions that can be applied, beginning at the start state, that lead to the given string.
  - When following such an approach, it is useful to think a few moves ahead.
  - This approach is known as **top-down parsing**.
- A second approach, known as **bottom-up parsing**, is to work backward from the given string with the goal of undoing productions one-by-one to reach the start symbol

## Parsing Example

Example 5: Determine whether the word *cbab* belongs to the language generated by the grammar $G = (V, T, S, P)$, where $V = \{a, b, c, A, B, C, S\}$, $T = \{a, b, c\}$, $S$ is the starting symbol, and the productions are

- $S \rightarrow AB$
- $A \rightarrow Ca$
- $B \rightarrow Ba$
- $B \rightarrow Cb$
- $B \rightarrow b$
- $C \rightarrow cb$
- $C \rightarrow b$.

## Grammars Summary

- Grammars are used to generate the words of a language and to determine whether a word is in a language.
- Formal languages, which are generated by grammars, provide models both for natural languages, such as English, and for programming languages, such as Python, C, and Java.
- In particular, grammars are extremely important in the construction and theory of compilers.

## 2. Finite-State Machines with Output

- Many kinds of machines, including computers, can be modeled using a structure called a **finite-state machine** (or **finite automaton**).
- A finite-state machine consists of a finite set of states, with a designated start state, an input alphabet, and a transition function that assigns a next state to every state and input pair.
- Finite-state machines are used in many diverse applications, including spell-checking programs, grammar checking, indexing, searching large bodies of text, speech recognition, XML, HTML, and network protocols.

## A Vending Machine Model

A vending machine accepts 5, and 10 Baht coins. Each item costs 20 bahts.

- When a total of 21 baht or more has been deposited, the machine returns the excess amount.
- The customer can push a orange button to receive a container of Orange Fanta, or a red button to receive a container of Coke.

## A Vending Machine Model

The machine can be in any of 8 different states $s_i$, $i$ = 0, 1, 2, 3, 4, where $s_i$ is the state where the machine has collected 5$i$ Bahts.

- The machine starts in state $s_0$ = 0 Bahts received.
- The possible inputs are 5 Bahts, 10 Bahts, the orange button ($O$), and the red button ($R$).
- The possible outputs are nothing ($n$), 5 Bahts, 10 Bahts, an orange Fanta, and a coke.

## Finite-State Machine Example

Example 1: Suppose that a student puts in a 5 Bahts followed by 2 10 Bahts, receive 5 Bahts back, and then push the orange button for orange Fanta.

1. The machine starts in state $s_0$
2. The 1st input is 5 Bahts: state $s_0$ -> $s_1$ and gives no output.
3. The 2nd input is 10 Bahts: state $s_1$ -> $s_3$ and gives no output.
4. The 3rd input is 10 Bahts: state $s_3$ -> $s_4$ and gives 5 Bahts as output.
5. The next input is orange button: $s_4$ -> $s_0$ and gives an orange Fanta as its output.

## Finite-State Machine Example

| | **Next State** | | | | **Output** | | | |
|---|---|---|---|---|---|---|---|---|
| | **Input** | | | | **Input** | | | |
| **State** | **5** | **10** | **O** | **R** | **5** | **10** | **O** | **R** |
| $s_0$ | $s_1$ | $s_2$ | $s_0$ | $s_0$ | n | n | n | n |
| $s_1$ | $s_2$ | $s_3$ | $s_1$ | $s_1$ | n | n | n | n |
| $s_2$ | $s_3$ | $s_4$ | $s_2$ | $s_2$ | n | n | n | n |
| $s_3$ | $s_4$ | $s_4$ | $s_3$ | $s_3$ | n | 5 | n | n |
| $s_4$ | $s_4$ | $s_4$ | $s_4$ | $s_4$ | 5 | 10 | Fanta | Coke |

## Finite-State Machine Example

## Finite-State Machines with Output

**Definition 1**: A **finite-state machine (FSM)** $\boldsymbol{M} = (S, I, O, f, g, s_0)$ consists of

- a finite set $S$ of ***states***, a finite ***input alphabet*** $I$,
- a finite ***output alphabet*** $O$,
- a **transition function** $f$ that assigns to each state and input pair a new state,
- an **output function** $g$ that assigns to each state and input pair an output,
- and an **initial state** $s_0$.

## Finite-State Machines with Output

- A **state table** is used to represent the values of the transition function *f* and the output function *g* for all pairs of states and input.

Example 2: A finite-state machine with $S = \{s_0, s_1, s_2, s_3\}$, $I = \{0, 1\}$, and $O = \{0, 1\}$.

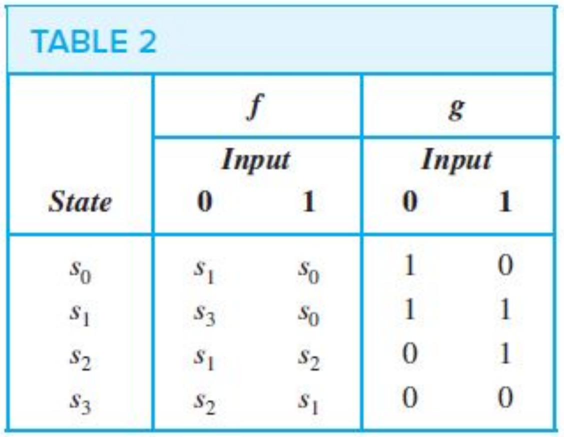
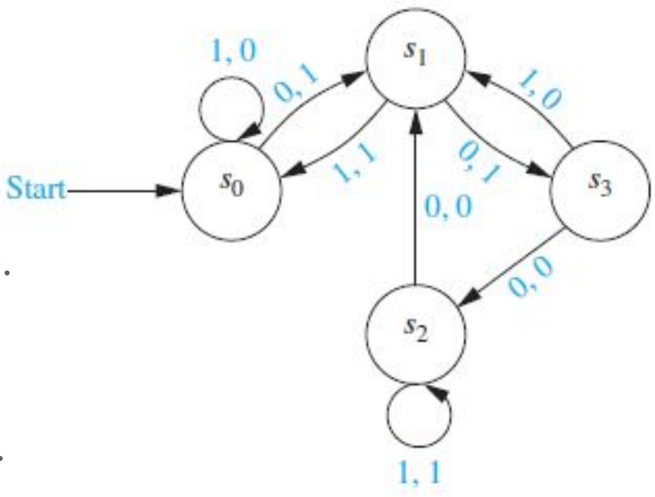


TABLE 2

| | *f* | | *g* | |
|---|---|---|---|---|
| | *Input* | | *Input* | |
| ***State*** | **0** | **1** | **0** | **1** |
| $s_0$ | $s_1$ | $s_0$ | 1 | 0 |
| $s_1$ | $s_3$ | $s_0$ | 1 | 1 |
| $s_2$ | $s_1$ | $s_2$ | 0 | 1 |
| $s_3$ | $s_2$ | $s_1$ | 0 | 0 |

## Finite-State Machines with Output

- Alternatively, a FSM can be represented by a **state diagram**, which is a directed graph with labeled edges.
  - Each state is represented by a circle.
  - Arrows labeled with the input and output pair represent the transitions.



## Finite-State Machine Exercise

Exercise 1: Construct the FSM based on the state table.

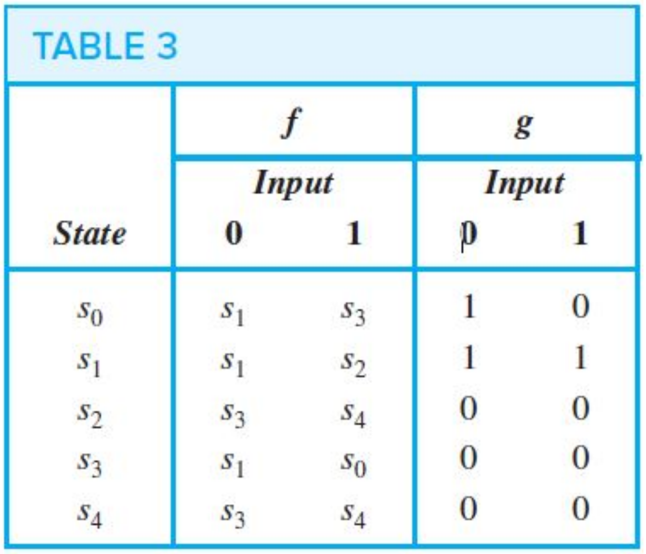


TABLE 3

| | $f$ | | $g$ | |
|---|---|---|---|---|
| | *Input* | | *Input* | |
| ***State*** | **0** | **1** | **0** | **1** |
| $s_0$ | $s_1$ | $s_3$ | 1 | 0 |
| $s_1$ | $s_1$ | $s_2$ | 1 | 1 |
| $s_2$ | $s_3$ | $s_4$ | 0 | 0 |
| $s_3$ | $s_1$ | $s_0$ | 0 | 0 |
| $s_4$ | $s_3$ | $s_4$ | 0 | 0 |

## Finite-State Machines with Output

- An input string (a sequence of inputs) takes the starting state through a sequence of states.
    - Each input symbol takes the machine from one state to another.
    - An input string also produces an output string.

## Finite-State Machines with Output

Example 2: Find the output string generated by the finite-state machine in Exercise 1 if the input string is 101011.

## Addition Machine

Example 3: Produce a finite-state machine that adds 2 positive integers using their binary expansions.

Recall the conventional procedure to add $(x_n \ldots x_1 x_0)_2$ and $(y_n \ldots y_1 y_0)_2$.

First, the bits $x_0$ and $y_0$ are added, producing a sum bit $z_0$ and a carry bit $c_0$.

Next the bits $x_1$ and $y_1$ are added together with the carry bit $c_0$.

This gives a sum bit $z_1$ and a carry bit $c_1$.

The procedure continues until the $n$th stage, where $x_n$, $y_n$ and the previous carry $c_{n-1}$ are added to produce the sum bit $z_n$ and the carry bit $c_n$, which is equal to the sum bit $z_{n+1}$.

## Addition Machine

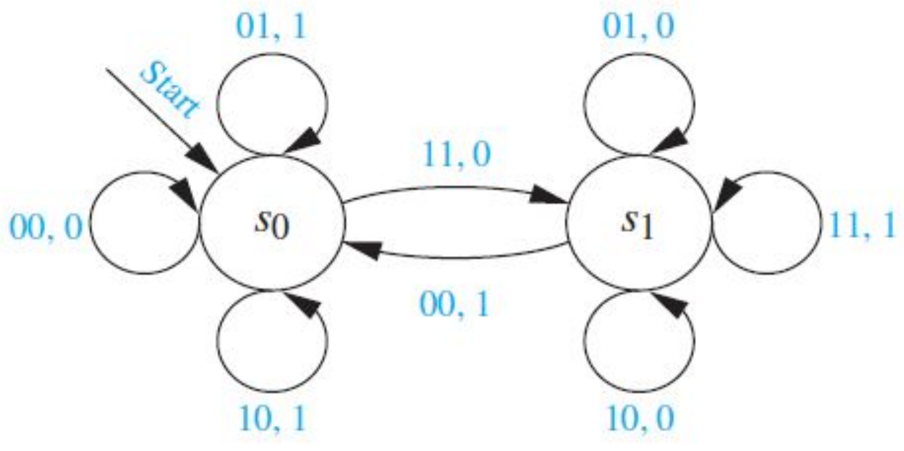
Example 3: Produce a finite-state machine that adds 2 positive integers using their binary expansions.

A FSM to carry out this addition can be constructed using just 2 states.

- The start state $s_0$ is used to remember that the previous carry is 0 (or for the addition of the rightmost bits).
- The other state, $s_1$, is used to remember that the previous carry is 1.
    - For simplicity we assume that both the initial bits $x_n$ and $y_n$ are 0.
- Because the inputs to the machine are pairs of bits, there are 4 possible inputs.

## Addition Machine

Example 3: Produce a finite-state machine that adds 2 positive integers using their binary expansions.

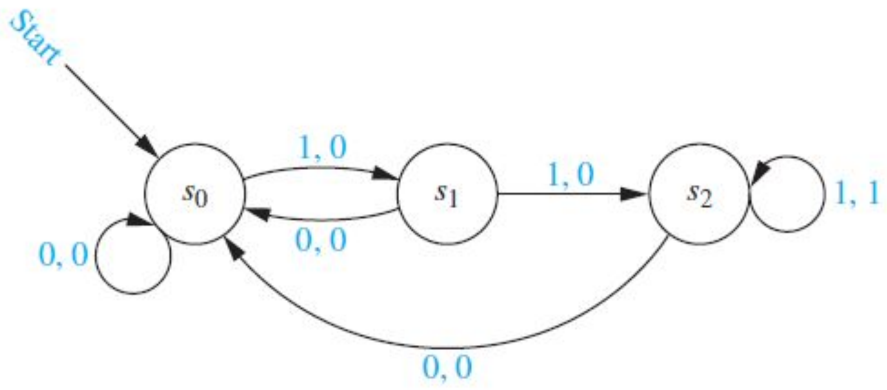


## FSM Exercise

Exercise 2: In a certain coding scheme, when three consecutive 1s appear in a message, the receiver of the message knows that there has been a transmission error. Construct a finite-state machine that gives a 1 as its current output bit if and only if the last three bits received are all 1s.

## FSM Exercise

Exercise 2: In a certain coding scheme, when three consecutive 1s appear in a message, the receiver of the message knows that there has been a transmission error. Construct a finite-state machine that gives a 1 as its current output bit if and only if the last three bits received are all 1s.



## Finite-State Machines Recognition

- The final output bit of the finite-state machine constructed is 1 if and only if the input string ends with 111.
    - Because of this, we say that this finite-state machine **recognizes** the set of bit strings that end with 111.

**Definition 2**: Let $M = (S, I, O, f, g, s_0)$ be a finite-state machine and $L \subseteq I^*$. We say that M ***recognizes*** (or accepts) $L$ if an input string $x$ belongs to $L$ if and only if the last output bit produced by $M$ when given $x$ as input is a 1.

## 3. Finite-State Machines with No Output

- A finite-state machine with output can be used to recognize a language, by giving an output of 1 when a string from the language has been read and a 0 otherwise.
- However, there are other types of finite-state machines that are specially designed for recognizing languages.
    - Instead of producing output, these machines have final states.
    - A string is recognized if and only if it takes the starting state to one of these final states.

## Set of Strings

**Definition 1**: Suppose that $A$ and $B$ are subsets of $V^*$, where $V$ is a vocabulary. The ***concatenation*** of $A$ and $B$, denoted by $AB$, is the set of all strings of the form $xy$, where $x$ is a string in $A$ and $y$ is a string in $B$.

Example 1: Let $A = \{0, 11\}$ and $B = \{1, 10, 110\}$. Find $AB$ and $BA$.

- AB = {01, 010, 0110, 111, 1110, 11110}
- BA = {10, 111, 100, 1011, 1100, 11011}

From this definition, we can define $A^n$, for $n = 0, 1, 2, \ldots$.

$$A_0 = \{\lambda\}, \quad A_{n+1} = A^n A \text{ for } n = 0, 1, 2, \ldots$$

## Set of Strings

Example 2: Let A = {1, 00}. Find $A^n$ for n = 0, 1, 2, and 3.

## Kleene Closure

**Definition 2**: Suppose that $A$ is a subset of $V^*$. Then the ***Kleene closure*** of A, denoted by $A^*$, is the set consisting of concatenations of arbitrarily many strings from $A$. That is, $A^* = \bigcup_{k=0}^{\infty} A^k$.

Example 3: What are the Kleene closures of the sets A = {0}, B = {0, 1}, and C = {11}?.

## Finite-State Automata (FSA)

- Basically, a finite-state machine with no output is ***finite-state automata***.

**Definition 2**: A **finite-state automaton *M*** = $(S, I, f, s_0, F)$ consists of

- a finite set S of states,
- a finite input alphabet $I$,
- a transition function $f$ that assigns a next state to every pair of state and input (so that $f: S \times I \to S$),
- an initial or start state $s_0$, and
- a subset ***F*** of $S$ consisting of final (or ***accepting states***).

## Finite-State Automata (FSA)

- FSA do not produce output, but they do have a set of final states.
  - They recognize strings that take the starting state to a final state.
- FSAs can be represented using either state tables or state diagrams, in which final states are indicated with a double circle.

## Finite-State Automata (FSA)

Example 4: Construct the state diagram for the finite-state automaton *M* = ($S$, $I$, $f$, $s_0$, $F$), where $S = \{s_0, s_1, s_2, s_3\}$, $I = \{0, 1\}$, $F = \{s_0, s_3\}$, and the transition function $f$ is given in a table shown here.

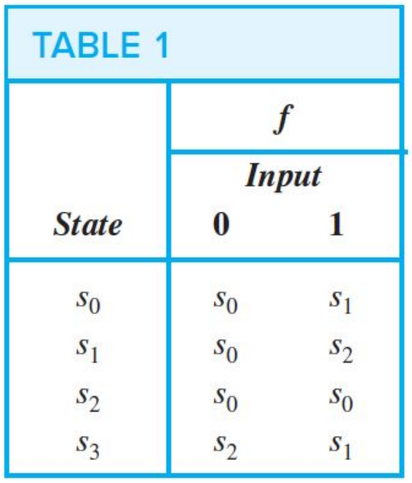


TABLE 1

| | ***f*** | |
|---|---|---|
| | ***Input*** | |
| ***State*** | **0** | **1** |
| $s_0$ | $s_0$ | $s_1$ |
| $s_1$ | $s_0$ | $s_2$ |
| $s_2$ | $s_0$ | $s_0$ |
| $s_3$ | $s_2$ | $s_1$ |

## Language Recognition by FSAs

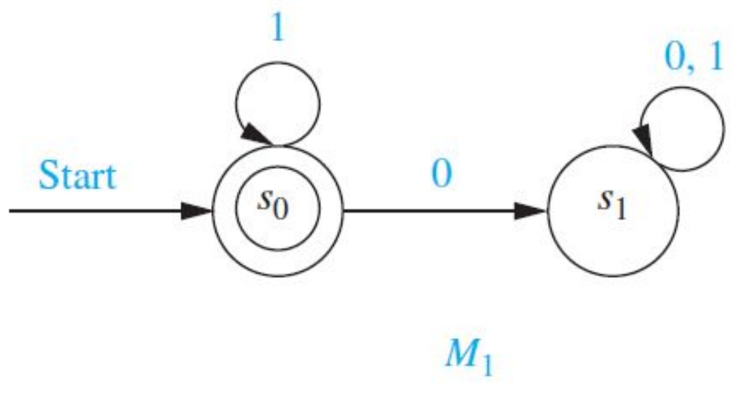
**Definition 3**: A string *x* is said to be ***recognized*** or ***accepted*** by the machine $M = (S, I, f, s_0, F)$ if it takes the initial state $s_0$ to a final state, that is, $f(s_0, x)$ is a state in *F*.

The ***language recognized*** or ***accepted*** by the machine M, denoted by ***L*(*M*)**, is the set of all strings that are recognized by M.

Two finite-state automata are called ***equivalent*** if they recognize the same language.

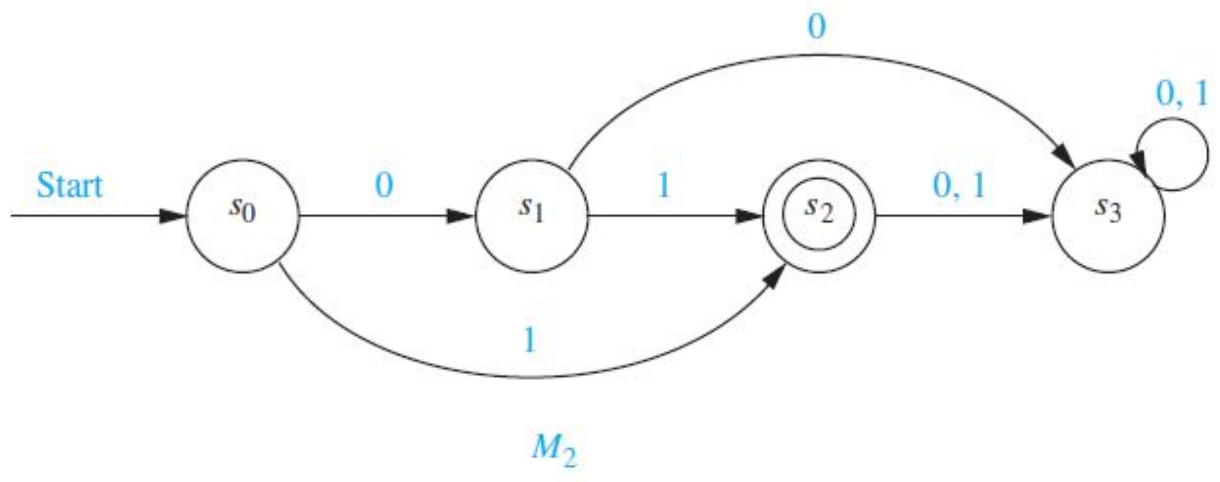
## Language Recognition by FSAs

Example 5: Determine the languages recognized by the FSA $M_1$ in a given figure.



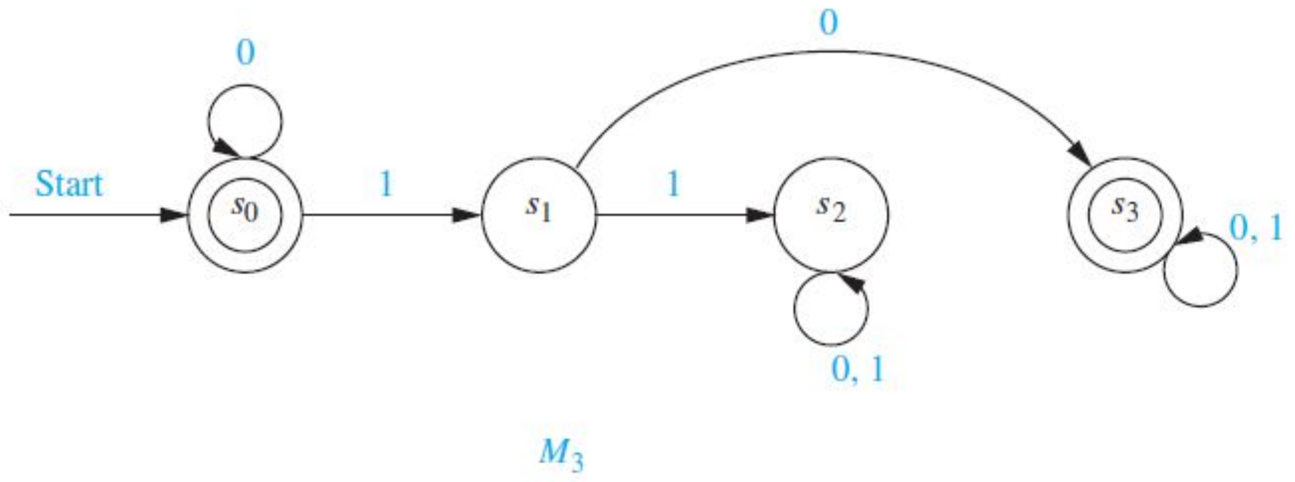
## Language Recognition by FSAs

Example 6: Determine the languages recognized by the FSA $M_2$ in a given figure.



## Language Recognition by FSAs

Example 7: Determine the languages recognized by the FSA $M_3$ in a given figure.



## Designing FSA

We can often construct a finite-state automaton that recognizes a given set of strings by

- carefully adding states and transitions
- determining which of these states should be starting state and final states.

When appropriate we include states that can keep track of some of the properties of the input string, providing the finite-state automaton with limited memory.

## Designing FSA Example

Example 8: Construct deterministic finite-state automata that recognize each of these languages.

(a) the set of bit strings that begin with two 0s

## Designing FSA Example

(b) the set of bit strings that contain two consecutive 0s

(c) the set of bit strings that do not contain two consecutive 0s

## Designing FSA Example

(d) the set of bit strings that end with two 0s

(e) the set of bit strings that contain at least two 0s

## Nondeterministic FSA (NDFSA)

- There is another important type of finite-state automaton in which there may be several possible next states for each pair of input value and state.
  - Such machines are called **nondeterministic**.

**Definition 4**: A nondeterministic finite-state automaton $M = (S, I, f, s_0, F)$ consists of

- a set $S$ of states,
- an input alphabet $I$,
- a transition function f that assigns a set of states to each pair of state and input (so that $f : S \times I \rightarrow P(S)$),
- a starting state $s_0$, and
- a subset $F$ of $S$ consisting of the final states.

## Nondeterministic FSA (NDFSA)

Example 9: Draw the state diagram for NDFSA with the state table shown. The final states are $s_2$ and $s_3$.

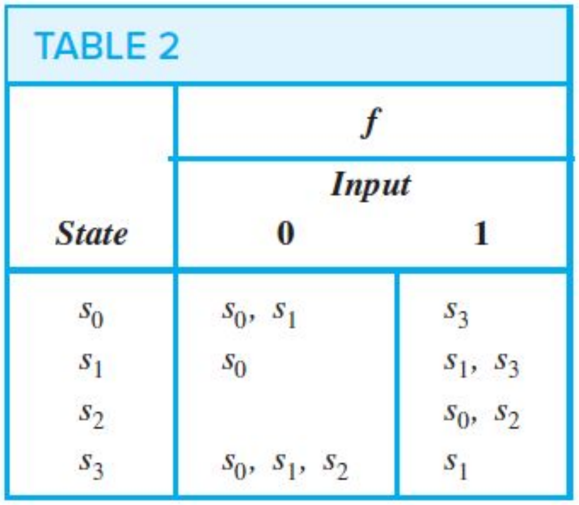
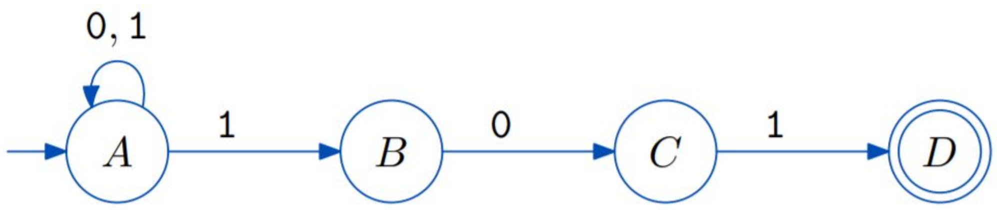


TABLE 2

| | *f* | |
|---|---|---|
| | *Input* | |
| *State* | **0** | **1** |
| $s_0$ | $s_0, s_1$ | $s_3$ |
| $s_1$ | $s_0$ | $s_1, s_3$ |
| $s_2$ | | $s_0, s_2$ |
| $s_3$ | $s_0, s_1, s_2$ | $s_1$ |

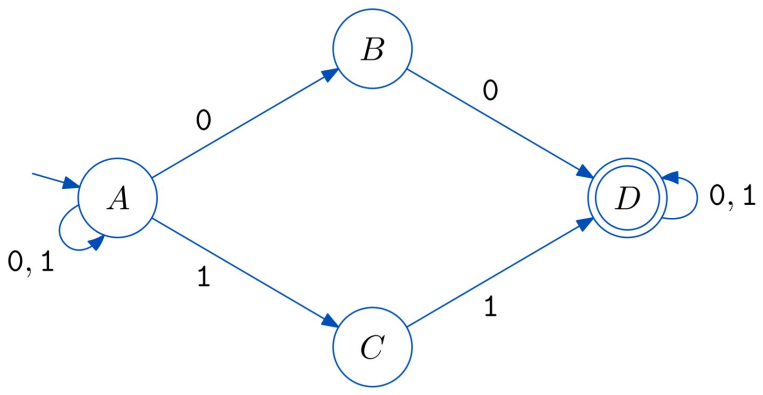
## Nondeterministic FSA (NDFSA)

Example 10: Find the language recognized $L(M)$ by the NDFSA shown in Figure.



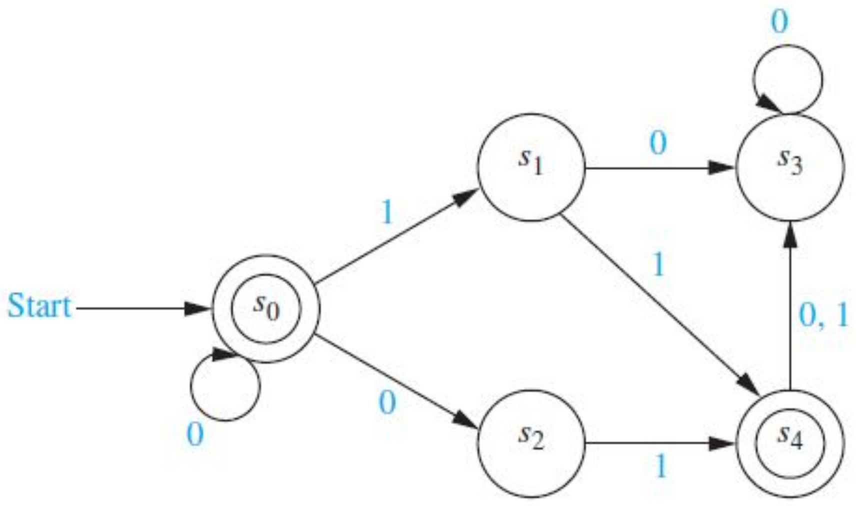
## Nondeterministic FSA (NDFSA)

Example 11: Find the language recognized $L(M)$ by the NDFSA shown in Figure.



## Nondeterministic FSA (NDFSA)

Example 12: Find the language recognized *L*(*M*) by the NDFSA shown in Figure.



## 4. Language Recognition

- FSA can be used as language recognizers.
- What sets can be recognized by these machines?
- American mathematician Stephen Kleene showed that there is a FSA that recognizes a set if and only if this set can be built up from the null set, the empty string, and singleton strings by taking concatenations, unions, and Kleene closures, in arbitrary order.
    - Sets that can be built up in this way are called **regular sets**.
    - A regular set is generated by a regular grammar.

## Regular Expressions

- To define regular sets, need to define regular expressions first.

Definition 1: The ***regular expressions*** over a set *I* are defined recursively by:

the symbol ∅ is a regular expression;

the symbol $\lambda$ is a regular expression;

the symbol $x$ is a regular expression whenever $x \in I$;

the symbols (**AB**), (**A** ∪ **B**), and **A*** are regular expressions whenever **A** and **B** are regular expressions.

## Regular Expressions

Each regular expression represents a set specified by these rules:

∅ represents the empty set, that is, the set with no strings;

$\boldsymbol{\lambda}$ represents the set $\{\lambda\}$, which is the set containing the empty string;

$x$ represents the set $\{x\}$ containing the string with one symbol $x$;

(**AB**) represents the concatenation of the sets represented by **A** and by **B**;

(**A** ∪ **B**) represents the union of the sets represented by **A** and by **B**;

**A*** represents the Kleene closure of the set represented by **A**.

- Sets represented by regular expressions are called **regular sets**.

## Regular Expressions Example

Example 1: What are the strings in the regular sets specified by the regular expressions

**10***

(**10**)*

**0** ∪ **01**

**0**(**0** ∪ **1**)*

(**0*****1**)*

## Regular Expressions Example

Example 2: Find a regular expression that specifies each of these set:

a) The set of bit strings with even length

b) The set of bit strings ending with a 0 and not containing 11

c) The set of bit strings containing an odd number of 0s

## Kleene’s Theorem

- In 1956, Kleene established connection between regular sets and sets recognized by a finite-state automaton.

**Kleene’s Theorem**: A set is regular if and only if it is recognized by a finite-state automaton.

## Regular Set and NDFSA

Example 3: Construct a NDFSA that recognizes the regular set $\mathbf{1}^* \cup \mathbf{01}$

## Finite-state Machines Summary

- All finite-state machines have a set of states, including a starting state, an input alphabet, and a transition function that assigns a next state to every pair of a state and an input.
- The states of a finite-state machine give it limited memory capabilities.
- Some finite-state machines produce an output symbol for each transition;
    - These machines can be used to model many kinds of machines, including vending machines, binary adders, and language recognizers.

## More Powerful Types of Machines

The main limitation of finite-state automata is their finite amount of memory. This has led to the development of more powerful models of computation.

- **Pushdown automaton** (PDA): includes a stack, which provides unlimited memory. It can recognize context-free grammar.
    - A set is recognized in one of the 2 ways
        - If the set consists of all the strings that produce an empty stack when they are used as input.
        - If the set consists of all the strings that lead to a final state when used as input.

## More Powerful Types of Machines

- **Linear Bounded Automata** (LBA): More powerful than PBA. It can recognize context-sensitive languages.
- **Turing Machine** (TM): Includes everything in a finite-state machine with a tape, which is infinite in both directions.
    - It can recognize all languages generated by phrase-structure grammars.

## 5. Turing Machines

- The finite-state automata studied earlier cannot be used as general models of computation. They are limited in what they can do.
- Basically, a Turing machine consists of a ***control unit,*** which at any step is in one of finitely many different states, together with a **tape** divided into cells, which is infinite in both directions.
- Turing machines have read and write capabilities on the tape as the control unit moves back and forth along this tape, changing states depending on the tape symbol read.

## Turing Machines

- Turing machines are more powerful than finite-state machines because they include additional memory capability.
- Turing machines are the most general models of computation; essentially they can do whatever a computer can do.
    - Note that Turing Machines are much more powerful than real computers, which have finite memory capabilities.

## Definition of Turing Machines

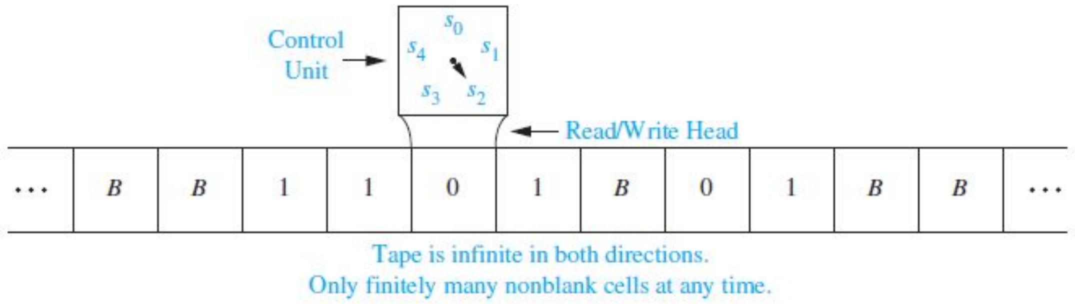
A Turing machine $T = (S, I, f, s_0)$ consists of

- a finite set $S$ of states,
- an alphabet $I$ containing the blank symbol $B$,
- a partial function $f$ from $S \times I$ to $S \times I \times \{R, L\}$, and
- a starting state $s_0$.

For some (state, symbol) pairs, the partial function $f$ maybe undefined, but for a pair for which it is defined, there is a unique (state, symbol, direction) triple associated to this pair.

The 5-tuples corresponding to the partial function in the definition of a TM are called the **transition rules** of the machine.

## Definition of Turing Machines


Control Unit
$s_0$
$s_1$
$s_2$
$s_3$
$s_4$
Read/Write Head
...
B
B
1
1
0
1
B
0
1
B
B
...
Tape is infinite in both directions.
Only finitely many nonblank cells at any time.

At each step, the control unit reads the current tape symbol *x*. If the control unit is in state *s* and if the partial function *f* is defined for the pair (*s*, *x*) with *f*(*s*, *x*) = (*s*′, *x*′, *d*), the control unit:

- enters the state *s*′,
- writes the symbol *x*′ in the current cell, erasing *x*, and
- moves right one cell if *d* = *R* or moves left one cell if *d* = *L*.

## Definition of Turing Machines

This step is written as the 5-tuple (*s*, *x*, *s*′, *x*′, *d*).

- Turing machines are defined by specifying a set of such 5-tuples.

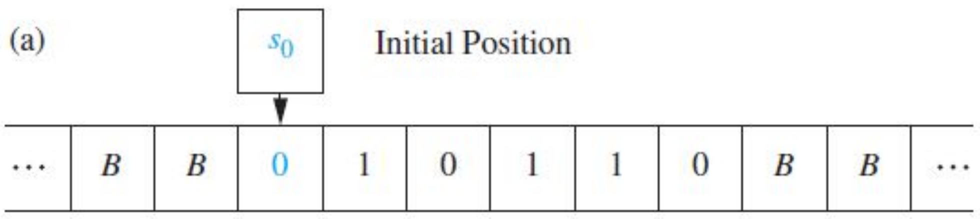
If the partial function *f* is undefined for the pair (*s*, *x*), then the Turing machine *T* will ***halt***.

At the beginning of its operation, a TM is assumed to be in the initial state $s_0$ and to be positioned over the leftmost nonblank symbol on the tape. This is the ***initial position*** of the machine.

If the tape is all blank, the control head can be positioned over any cell.

## A TM in Operation

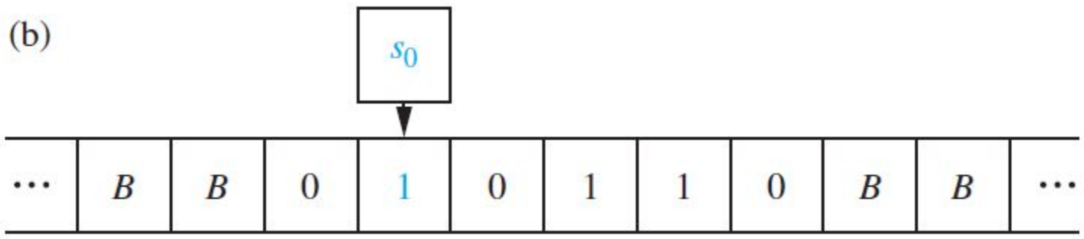
Example 1: What is the final tape when the Turing machine *T* defined by the seven 5-tuples $(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 1, R)$, $(s_0, B, s_3, B, R)$, $(s_1, 0, s_0, 0, R)$, $(s_1, 1, s_2, 0, L)$ $(s_1, B, s_3, B, R)$, and $(s_2, 1, s_3, 0, R)$ is run on the tape shown in Figure?



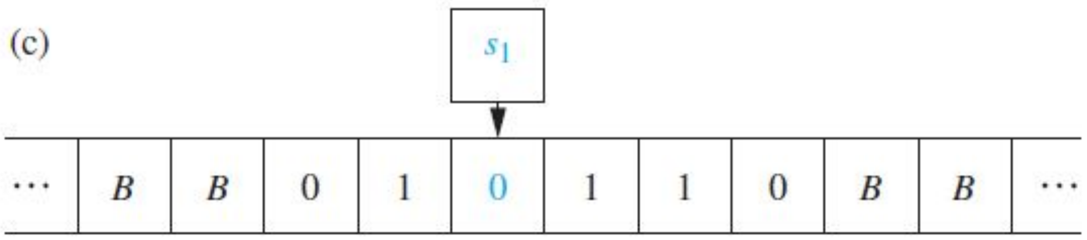
Start the operation with T in state $s_0$ and with *T* positioned over the leftmost nonblank symbol on the tape. The first step, using the 5-tuple $(s_0, 0, s_0, 0, R)$, reads the 0 in the leftmost nonblank cell, stays in state $s_0$, writes a 0 in this cell, and moves one cell right.

## A TM in Operation

$(s_0, 1, s_1, 1, R)$



$(s_1, 0, s_0, 0, R)$



## A TM in Operation

## A TM in Operation

## Using Turing Machines to Recognize Sets

A ***final state*** of a Turing machine $T$ is a state that is not the first state in any 5-tuple in the description of $T$ using 5-tuples (for example, state $s_3$ in last example).

**Definition 2:** Let $V$ be a subset of an alphabet $I$. A Turing machine $T = (S, I, f, s_0)$ **recognizes** a string $x$ in $V^*$ if and only if $T$, starting in the initial position when $x$ is written on the tape, halts in a final state.

- $T$ is said to recognize a subset $A$ of $V^*$ if $x$ is recognized by $T$ if and only if $x$ belongs to $A$.
- A TM operating on a tape containing the symbols of a string $x$ in consecutive cells, **does not recognize** $x$ if it does not halt or halts in a state that is not final.

## Using Turing Machines to Recognize Sets

Example 2: Find a TM that recognizes the set of bit strings that have a 1 as their second bit, that is, the regular set $(\mathbf{0} \cup \mathbf{1})\mathbf{1}(\mathbf{0} \cup \mathbf{1})^*$

We want a Turing machine that, starting at the leftmost nonblank tape cell, moves right, and determines whether the second symbol is a 1.

- If the second symbol is 1, the machine should move into a final state.
- If the second symbol is not a 1, the machine should not halt or it should halt in a nonfinal state.

## Using Turing Machines to Recognize Sets

Example 2: Find a TM that recognizes the set of bit strings that have a 1 as their second bit, that is, the regular set $(\mathbf{0} \cup \mathbf{1})\mathbf{1}(\mathbf{0} \cup \mathbf{1})^*$

## Computing Functions with Turing Machines

- A Turing machine can be thought of as a computer that finds the values of a partial function.
- Suppose that the TM $T$, when given the string $x$ as input, halts with the string $y$ on its tape. W can then define $T(x) = y$. The domain of $T$ is the set of strings for which $T$ halts.

## Computing Functions with Turing Machines

- How can we use Turing machines to compute functions defined on integers, on pairs of integers, on triples of integers, and so on?
- To consider a TM as a computer of functions from the set of *k*-tuples of nonnegative integers to the set of nonnegative integers (such functions are called **number-theoretic** functions), we need a way to represent *k*-tuples of integers on a tape.
- To do so, we use **unary representations** of integers.

## Computing Functions with Turing Machines

- We represent the nonnegative integer *n* by a string of *n* + 1 1s
  - So that, for instance, 0 is represented by the string 1
  - 5 is represented by the string 111111.
- To represent the *k*-tuple $(n_1, n_2, \ldots, n_k)$, we use a string of $n_1 + 1$ 1s, followed by an asterisk, followed by a string of $n_2 + 1$ 1s, followed by an asterisk, and so on, ending with a string of $n_k + 1$ 1s.
  - For example, to represent the 4-tuple (2, 0, 1, 3)
    - we use the string $111 * 1 * 11 * 1111$.

## Computing Functions Example

Example 3: Construct a Turing machine for adding 2 nonnegative integers.

We need to build a Turing machine *T* that computes the function $f(n_1, n_2) = n_1 + n_2$. The pair $(n_1, n_2)$ is represented by a string of $n_1 + 1$ 1s followed by an asterisk followed by $n_2 + 1$ 1s.

The machine *T* should take this as input and produce as output a tape with $n_1 + n_2 + 1$ 1s.

## Turing Machine Summary

- Turing machines can be used to recognize sets, compute number-theoretic functions.
- The **Church–Turing thesis** states that every effective computation can be carried out using a Turing machine.
- Constructing Turing machines to compute relatively simple functions can be extremely demanding.

## Turing Machine Summary

Certainly, though, any problem that can be solved using a computer with a program written in any language, perhaps using an unlimited amount of memory, should be considered effectively **solvable**.

- Note that Turing machines have unlimited memory, unlike computers in the real world, which have only a finite amount of memory.

Turing machines can be used to study the difficulty of solving certain classes of problems.

    - In particular, Turing machines are used to classify problems as tractable versus intractable and solvable versus unsolvable.