3. Logic Thinking

> Computer science is the continuation of logic by other means.
>
> Georg Gottlob, 2009

Logic thinking is concerned with one of the key questions in computer science: what kind of problems can be *correctly* solved by computational processes? This question can be broken down into two issues:

- The **correctness** issue. How to rigorously define correctness of computational processes? In doing so, we can have a rigorous definition of computable problems: those problems for which there exist correct computational processes.
- The **generality** issue. Is there a computer that can be used to solve any computable problem?

We introduce two bit-accurate models of computation: Boolean logic and Turing machine, to rigorously define and analyze the correctness and generality of computational processes. Four main points are emphasized:

- With Boolean logic and Turing machine, we are able to accurately model a problem and the computational process to solve the problem. We can also define and verify the correctness of the solution.
- The above method is universal, that is, Boolean logic and Turing machine can be used to model all computable problems and their solutions.
- Boolean logic and Turing machines have limitations. There exist undecidable problems, paradoxes and incompleteness theorems.
- Logic thinking in computer science has differences from logic thinking of other sciences. Logic thinking in computer science emphasizes bit accuracy and automatic execution.

We proceed in the following sequence.

- To ensure the correctness of a computational process, we make sure that each step of the process is correct and that compositions of steps are correct.
- To ensure one step is correct, we use Boolean logic.
- To ensure the correctness of multiple steps, we use Turing machines.
- To see the power and limitation of Turing machines, we study Church-Turing Hypothesis and Gödel's incompleteness theorems.

3.1. Boolean Logic

Boolean logic is a formal logic system to reason about logic statements which can have true (1) or false (0) values. Boolean logic has manifested in propositional logic and predicate logic. It is a perfect match for computer science, because computers use binary values of 0 and 1 to represent digital symbols.

We first use three examples to illustrate what problems can be solved by Boolean logic. Solutions to these problems will be provided in later examples.

- The Congruent Triangles Problem, to show that Boolean logic can be used to solve mathematic problems, without using mathematic domain knowledge.
- The Impatient Guide Problem, to illustrate that problems in many application domains can be encoded as Boolean logic problems.
- The Adder Implementation Problem, to show that many computer hardware components can be implemented as Boolean logic expressions.

Example 12. The congruent triangles problem

Let us consider a statement in geometry: *congruent triangles are also similar*. More precisely, if two triangles are congruent, they are also similar.

We are taught in geometry class that this statement is true. The teacher may even have shown us a proof, using geometry knowledge.

Now consider another statement, which is related to the original statement:

If two triangles are not similar, then they are not congruent.

Logic thinking can be used to show that the second statement holds. The proof is very simple. More importantly, it does not involve any knowledge in geometry.

☶

Example 13. The impatient guide problem

A tourist is traveling in the land of Oz and wants to go to the Emerald City. The tourist reaches a crossroad with paths P and Q, one of which leads to the Emerald City. There is a guide G at the crossroad, who comes from either the Honest Village or the Lying Village. Anyone from the Honest Village always tells the truth, and anyone from the Lying Village always tells lies. The guide is impatient, in that G only answers one question from the tourist, and the answer is either "Yes" or "No".

What question should the tourist ask the guide, to determine the correct path?

☶

Example 14. The adder implementation problem

Realizing addition of two *n*-bit numbers with Boolean logic. This book asks students to realize adders in several contexts, because adders are simple and fundamental. If we can do addition, we can also do many other operations.

☶

3.1.1. Propositional Logic

Propositional logic is a logic system for reasoning about combinational propositions made of propositional variables and logic connectives. An example is the proposition "If the Earth is flat, then Alan Turing is a computer scientist". In this section, we will learn how to understand such a proposition and how to decide whether it is true or false.

10. Propositions and logic connectives

A **proposition** is a declarative sentence which has a truth-value, that is, being true or false. For example, "this book is written in English"; "Beijing is China's capital city". A proposition can contain one or more other propositions as parts. For example, "5 is a prime number *and* $5 \equiv 1 \pmod 4$ ".

We use variables x, y, z to denote proposition, and $x = 1$ means proposition x is true while $x = 0$ means proposition x is false.

The word "and" in the previous example is a *logic connective* called *conjunction*. Propositional logic largely involves studying these kinds of logical connectives and the rules determining the truth-values of the propositions combined from simpler propositions and collectives. Five logical connectives are commonly used. Their definitions and illustrative examples are shown in Box 7.

Four of the five connectives are straightforward. The implication connective may look strange for beginners to logic. A key observation is that a false premise implies anything! Thus, both of the following propositions are true:

The Earth is flat → Alan Turing is a computer scientist
The Earth is flat → Alan Turing is not a computer scientist

Box 7. Commonly Used Logic Connectives

- **Conjunction** ∧ (also called **AND**): $x \wedge y = 1$ if and only if $x = y = 1$.
  For example, the solution of $x^2 + 2x < 0$ is $x > -2$ and $x < 0$. We can describe it as $(x > -2) \wedge (x < 0)$.
- **Disjunction** ∨ (also called **OR**): $x \vee y = 0$ if and only if $x = y = 0$.
  For example, the solution of $x^2 + 2x \geq 0$ satisfies $x \leq -2$ or $x \geq 0$. We can describe it as $(x \leq -2) \vee (x \geq 0)$.
- **Negation** ¬ (also called **NOT**): $\neg x = 0$ if and only if $x = 1$.
  We also use $\bar{x}$ to represent the negation of x, that is, $\bar{x} = \neg x$.
- **Implication** →: $(x \to y) = 1$ if and only if $x = 0$ or $y = 1$.
  We call x **premise** and y **conclusion**. Implication means $x \to y = 1$ if and only if the premise is false or the conclusion is true.
- **Exclusive-or** (also called **XOR**) $\oplus$: $x \oplus y = 1$ if and only if $x \neq y$.
  That is, $x \oplus y$ is true iff either x or y (but not both) is true. Note $x \oplus 1 = \neg x$, that is, we can use exclusive-or operation to realize negation.

11. Truth table

For any proposition, we can list its truth values on all of the possible combinations of values of the variables, to form its truth table. The following table lists the truth values for conjunction, disjunction, implication, and exclusive-or operations, given the combinations of the truth values of propositional variables *x* and *y*.

Table 3.1 Truth table for conjunction, disjunction, implication, and exclusive-or

x	y	$x \wedge y$	$x \vee y$	$x \rightarrow y$	$x \oplus y$
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	1	0

12. Properties of logic connectives

We call a proposition without connectives a *primitive proposition*, and a proposition with one or more connectives a *combinational proposition*. The truth value of a primitive proposition is not decided by proposition logic, but by the environment or context where the primitive proposition is made. Proposition logic is concerned with the truth values of combinational propositions, given the truth values of their primitive propositions and the combinations with logic connectives.

From definitions of logic connectives in Box 7, we can derive basic properties of propositional logic, as listed in Table 3.2. Additional properties are listed in Table 3.3. We omit the proof and the reader can use truth table to verify these properties.

A simple way to learn these properties is to contrast them to the logic we learned from high-school math or algebra classes. Some basic properties of propositional logic in Table 3.2 do not hold in high-school logic, shown in red. Note that we use the more familiar addition symbol + and multiplication symbol • from school classes, to denote disjunction ($\vee$) and conjunction ($\wedge$).

The associativity, commutativity, and identity laws hold for both propositional logic and high school mathematical logic. However, the distributivity and the annihilator laws only hold for multiplication but not addition. In high school math, multiplication distributes over addition, e.g., $(2 + 3) \cdot 4 = (2 \cdot 4) + (3 \cdot 4)$, but not addition distributed over multiplication, e.g., $(2 \cdot 3) + 4 \neq (2 + 4) \cdot (3 + 4)$. The other three laws of idempotence, absorption, and complementation do not hold at all in high school math.

Table 3.2 Basic properties of propositional logic

Law	Logic Equivalence	School Logic (assume x=2, y=3, z=4)
Associativity	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$,	$(2 \cdot 3) \cdot 4 = 2 \cdot (3 \cdot 4)$
	$(x + y) + z = x + (y + z)$	$(2 + 3) + 4 = 2 + (3 + 4)$
Commutativity	$x \cdot y = y \cdot x$	$2 \cdot 3 = 3 \cdot 2$
	$x + y = y + x$	$3 + 2 = 2 + 3$
Distributivity	$(x + y) \cdot z = (x \cdot z) + (y \cdot z)$	$(2 + 3) \cdot 4 = (2 \cdot 4) + (3 \cdot 4)$
	$(x \cdot y) + z = (x + z) \cdot (y + z)$	$(2 \cdot 3) + 4 \neq (2 + 4) \cdot (3 + 4)$
Identity	$x + 0 = x$, $x \cdot 1 = x$	$2 + 0 = 2$, $2 \cdot 1 = 2$
Annihilator	$x \cdot 0 = 0$, $x + 1 = 1$	$2 \cdot 0 = 0$, $2 + 1 \neq 1$
Idempotence	$x \cdot x = x$, $x + x = x$	$2 \cdot 2 \neq 2$, $2 + 2 \neq 2$
Absorption	$(x \cdot y) + x = x$, $(x + y) \cdot x = x$	$(2 \cdot 3) + 2 \neq 2$, $(2 + 3) \cdot 2 \neq 2$
Complementation	$x + \neg x = 1$, $x \cdot \neg x = 0$	N/A

Table 3.3 lists additional properties for propositional logic which mainly consider negation, implication, and exclusive-or operations.

Table 3.3 Additional properties of propositional logic

Law	Logic Equivalence
De Morgan law	$\overline{x \wedge y} = \bar{x} \vee \bar{y}$, $\overline{x \vee y} = \bar{x} \wedge \bar{y}$
Implication defined in $\vee, \neg$	$x \rightarrow y = \bar{x} \vee y$.
XOR defined in $\wedge, \vee, \neg$	$x \oplus y = (\bar{x} \wedge y) \vee (x \wedge \bar{y})$, $x \oplus y = (x \vee y) \wedge (\bar{x} \vee \bar{y})$
Associativity for XOR	$x \oplus (y \oplus z) = (x \oplus y) \oplus z$
Commutativity for XOR	$x \oplus y = y \oplus x$
Identity for XOR	$x \oplus 0 = x$, $x \oplus 1 = \bar{x}$

13. Boolean expression and Boolean function

When we view logic connectives as operators, primitive and combinational propositions will become Boolean expressions. More formally, the set L of all **Boolean expressions** is recursively defined as follows.

1. Initially, let $0, 1, x_1, \ldots, x_n \in L$, where $x_1, \ldots, x_n$ are primitive propositions for some natural number n. We also call 0 and 1 *Boolean constants*, and $x_1, \ldots, x_n$ *Boolean variables*.
2. Recursively apply negation, conjunction, disjunction, implication, and exclusive-or operations: If $x, y \in L$, then $\neg x, x \cdot y, x + y, x \rightarrow y, x \oplus y \in L$. Use the basic properties in Table 3.2 or 3.3 (sometimes called **Boolean algebra axioms**) to reduce all equivalent expressions into one expression.
3. Repeat Step 2 until L does not change any more.

We also have the notion of Boolean functions. An n-input-1-output **Boolean function** is a mathematical function $f:\{0,1\}^n \to \{0,1\}$, and the mapping is defined by the truth table of f. For instance, the equation $y = x_1 \oplus x_2 \oplus \cdots \oplus x_n$ defines a Boolean function to find the parity of $x_1, x_2, \cdots, x_n$, where the output is y and the n inputs are $x_1, x_2, \cdots, x_n$.

14. Normal forms

Given any Boolean expression, we can use the basic properties of Tables 3.2 and 3.3 to find an equivalent expression which contains only AND, OR, and NOT. For example, $(\mathrm{x} \vee \mathrm{y}) \to \mathrm{z} = (\overline{x \vee y}) \vee z = (\bar{x} \wedge \bar{y}) \vee z = (\bar{x} \vee z) \wedge (\bar{y} \vee z)$. Note that there are several different ways to present an expression with only AND, OR, and NOT. For example, the above example provides three different ways to represent $(\mathrm{x} \vee \mathrm{y}) \to \mathrm{z}$ with only AND, OR, and NOT. However, there is a uniform way to achieve this via the truth table. Let us write down the truth table for $(x \vee y) \to z$.

Table 3.4 Truth table for proposition $(x \vee y) \to z$

x	y	z	$(x \vee y) \to z$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

From the truth table, we know that there are three cases where the expression $(x \vee y) \to z$ is false: (1) $x = 0,\ y = 1,\ z = 0$; (2)$x = 1, y = z = 0$; (3) $x = y = 1,\ z = 0$. For the remaining five cases, the expression is true. For each case when the expression is true, we can use a product clause to represent it. For example, clause $\bar{x} \wedge \bar{y} \wedge \bar{z}$ represents the case $x = y = z = 0$, since $\bar{x} \wedge \bar{y} \wedge \bar{z} = 1$ if and only if $x = y = z = 0$. We then use $\vee$ to connect those true clauses to represent the whole expression:

$$(\bar{x} \wedge \bar{y} \wedge \bar{z}) \vee (\bar{x} \wedge \bar{y} \wedge z) \vee (\bar{x} \wedge y \wedge z) \vee (x \wedge \bar{y} \wedge z) \vee (x \wedge y \wedge z).$$

It is easy to check that the truth table for the above proposition is the same as the truth table for $(x \vee y) \to z$. Thus, they are equivalent propositions. We call it the **disjunctive normal form**. Actually, for any proposition, we can use the above method to write down its disjunctive normal form.

Theorem: For any proposition $F(x_1, x_2, \dots, x_n) \not\equiv 0$ with n variables, we can uniquely represent it as the following disjunctive normal form:

$$\boldsymbol{F(x_1, x_2, \dots, x_n) = Q_1 \vee Q_2 \vee \cdots \vee Q_m}$$

where each product $Q_i = l_1 \wedge l_2 \wedge \cdots \wedge l_n$, and $l_j = x_j$ or $\bar{x_j}$.

This theorem is obtained by looking at 1-valued rows in a truth table. If we look at 0-valued rows in the truth table, can we also write an expression for $(x \vee y) \rightarrow z$? The answer is YES. For each 0-valued row, we can write the representations connected by OR. For example, the row "$x = 0,\ y = 1,\ z = 0$" can be represented by $(x \vee \bar{y} \vee z)$. It means $(x \vee \bar{y} \vee z) = 0$ if and only if $x = 0,\ y = 1,\ z = 0$. Finally, we use AND to connect the representations for the three 0-valued rows.

$$(x \vee y) \rightarrow z = (x \vee \bar{y} \vee z) \wedge (\bar{x} \vee y \vee z) \wedge (\bar{x} \vee \bar{y} \vee z).$$

This is another way to write an equivalent proposition for any proposition, and we call it the **conjunctive normal form**.

Theorem: For any proposition $F(x_1, x_2, \dots, x_n) \not\equiv 1$ with n variables, we can uniquely represent it as the following conjunctive normal form:

$$\boldsymbol{F(x_1, x_2, \dots, x_n) = Q_1 \wedge Q_2 \wedge \cdots \wedge Q_m}$$

where each sum $Q_i = l_1 \vee l_2 \vee \cdots \vee l_n$, and $l_j = x_j$ or $\bar{x_j}$.

15. The number of Boolean functions

Given a natural number $n > 0$, how many different Boolean functions are there with n input variables?

Note that a Boolean function may be represented as two or more equivalent Boolean expressions. Two Boolean expressions are equivalent if they have the same truth table. This is the same as saying that two Boolean expressions are equivalent if they have the same disjunctive normal form or if they have the same conjunctive normal form. Two Boolean functions are different, if they do not have equivalent Boolean expressions. Thus, all equivalent Boolean expressions are counted as one, when computing the number of different Boolean functions.

The original question can be reduced to the question: how many different truth tables there are for n input variables and one output variable. Let us look at a truth table of n input variables in general.

x_1	x_2	…	x_{n-1}	x_n	y
0	0	…	0	0	0 or 1
0	0	…	0	1	0 or 1
0	0	…	1	0	0 or 1
0	0	…	1	1	0 or 1
…	…	…	…	…	0 or 1
1	1	…	1	0	0 or 1
1	1	…	1	1	0 or 1

Any truth table has 2^n rows. Consequently, the *y* column has 2^n cells, and each can have a 0 or 1 value. Each different configuration of 0/1 values in the 2^n cells represents a different truth table. There are 2^{2^n} configurations. Thus, there are 2^{2^n} truth tables. So, there are 2^{2^n} distinct Boolean functions.

The above result implies that any given Boolean function can be implemented by a Boolean expression. From the normal form theorems, any Boolean function can be implemented by AND, OR, NOT operations. This gives an affirmative answer to the Adder Implementation Problem, since Adder is a Boolean function.

☶

Example 15. The numbers of Boolean functions of one and two variables

For any given integer $n > 0$, there are 2^{2^n} distinct Boolean functions. Let us understand this result more concretely by explicitly enumerating all Boolean expressions for n=1 and n=2.

First consider the case when $n = 1$. The number of Boolean functions is 2^{2^1}=4. We can use the recursive definition of Boolean expressions to find all Boolean functions of one input variable *x*. The four functions are shown below with their truth tables. They are: *y* is always false, *y* is always true, $y = x$, and y = NOT x.

x	y
0	**1**
1	**1**

$y = 1$

x	y
0	**0**
1	**0**

$y = 0$

x	y
0	**0**
1	**1**

$y = x$

x	y
0	**1**
1	**0**

$y = \overline{x}$

Now consider the case when $n = 2$. The number of Boolean functions is 2^{2^2}=16. Again, we use the recursive definition of Boolean expressions to find all Boolean functions of two input variables x_1 and x_2.

Round 1. $y = 0,\ y = 1, y = x_1, y = x_2,\ y = \overline{x_1},\ y = \overline{x_2}$. Note that we simplify the process by putting negations of Boolean variables in the initial step. At the end of round 1, we have the Boolean expression set $L = \{0, 1, x_1, x_2, \overline{x_1},\ \overline{x_2}\}$. The corresponding truth tables are shown below.

x_1	x_2	y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**0**

$y = 0,$

x_1	x_2	y
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**1**

$y = 1,$

x_1	x_2	y
0	0	**0**
0	1	**0**
1	0	**1**
1	1	**1**

$y = x_1,$

x_1	x_2	y
0	0	**0**
0	1	**1**
1	0	**0**
1	1	**1**

$y = x_2$

x_1	x_2	y
0	0	**1**
0	1	**1**
1	0	**0**
1	1	**0**

$y = \overline{x_1}$,

x_1	x_2	y
0	0	**1**
0	1	**0**
1	0	**1**
1	1	**0**

$y = \overline{x_2}$

Round 2. Apply AND, OR, NOT to L, and use the axioms of Boolean expressions to eliminate redundant expressions. We add to L eight new expressions:

$\overline{x_1} \vee \overline{x_2}$, $\overline{x_1} \vee x_2$, $x_1 \vee \overline{x_2}$, $x_1 \vee x_2$, $\overline{x_1} \wedge \overline{x_2}$, $\overline{x_1} \wedge x_2$, $x_1 \wedge \overline{x_2}$, $x_1 \wedge x_2$

We have the new Boolean expression set

$L = \{0, 1, x_1, x_2, \overline{x_1}, \overline{x_2}; \overline{x_1} \vee \overline{x_2}, \overline{x_1} \vee x_2, x_1 \vee \overline{x_2}, x_1 \vee x_2, \overline{x_1} \wedge \overline{x_2}, \overline{x_1} \wedge x_2, x_1 \wedge \overline{x_2}, x_1 \wedge x_2\}$.

The corresponding truth tables for the eight new expressions are shown below.

x_1	x_2	y
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

$y = \overline{x_1} \vee \overline{x_2}$,

x_1	x_2	y
0	0	**1**
0	1	**1**
1	0	**0**
1	1	**1**

$y = \overline{x_1} \vee x_2$,

x_1	x_2	y
0	0	**1**
0	1	**0**
1	0	**1**
1	1	**1**

$y = x_1 \vee \overline{x_2}$,

x_1	x_2	y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

$y = x_1 \vee x_2$,

x_1	x_2	y
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

$y = \overline{x_1} \wedge \overline{x_2}$,

x_1	x_2	y
0	0	**0**
0	1	**1**
1	0	**0**
1	1	**0**

$y = \overline{x_1} \wedge x_2$,

x_1	x_2	y
0	0	**0**
0	1	**0**
1	0	**1**
1	1	**0**

$y = x_1 \wedge \overline{x_2}$,

x_1	x_2	y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

$y = x_1 \wedge x_2$

Round 3. Apply AND, OR, NOT to L, and use the axioms of Boolean expressions to eliminate redundant expressions. We add to L two new expressions:

$(\overline{x_1} \wedge \overline{x_2}) \vee (x_1 \wedge x_2)$, $(\overline{x_1} \wedge x_2) \vee (x_1 \wedge \overline{x_2})$

We have the new Boolean expression set

$L = \{0, 1, x_1, x_2, \overline{x_1}, \overline{x_2}; \overline{x_1} \vee \overline{x_2}, \overline{x_1} \vee x_2, x_1 \vee \overline{x_2}, x_1 \vee x_2, \overline{x_1} \wedge \overline{x_2}, \overline{x_1} \wedge x_2, x_1 \wedge \overline{x_2}, x_1 \wedge x_2; (\overline{x_1} \wedge \overline{x_2}) \vee (x_1 \wedge x_2), (\overline{x_1} \wedge x_2) \vee (x_1 \wedge \overline{x_2})\}$.

The corresponding truth tables for the two new expressions are shown below.

x_1	x_2	y
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

x_1	x_2	y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

$y = (\overline{x_1} \wedge \overline{x_2}) \vee (x_1 \wedge x_2)$, $y = (\overline{x_1} \wedge x_2) \vee (x_1 \wedge \overline{x_2})$

Round 4. Apply AND, OR, NOT to *L* to get no more new expressions. Stop. The final set of all 16 Boolean expressions are in the *L* obtained in Round 3.

☶

Example 16. The adder implementation problem, revisited

Students are asked to implement an adder, which takes two *n*-bit numbers *X* and *Y* as inputs and produce an *n*-bit number *Z* as the output. The adder is an *n*-input-*n*-output Boolean function. Since any *n*-input-1-output Boolean function can be implemented by an *n*-variable Boolean expression, we can theoretically use *n* such Boolean expressions to implement an *n*-input-*n*-output Boolean function. In practice, we can often have better implementations.

Let us start at *n*=1. A **full adder** has three input variables x_1, y_1, c_0 and two output variables z_1, c_1. Sometimes we simplify the variables as x, y, c_{in}, z, c_{out}, where $c_{in} = c_0$ is the carry-in, and $c_{out} = c_1$ is the carry-out. The function of a full adder can be understood as $x_1 + y_1 + c_{in} = (c_{out}z)_2$ where x_1, y_1, c_{in} are three 1-bit binary numbers and $(c_{out}z)_2$ is a 2-bit binary number. We can use 2 Boolean expressions to compute z, c_{out}. See the following equations and truth table:

$$z = x \oplus y \oplus c_{in}; \quad c_{out} = (x \wedge y) \vee ((x \oplus y) \wedge c_{in})$$

c_{in}	x	y	z	c_{out}
0	0	0	**0**	**0**
0	0	1	**1**	**0**
0	1	0	**1**	**0**
0	1	1	**0**	**1**
1	0	0	**1**	**0**
1	0	1	**0**	**1**
1	1	0	**0**	**1**
1	1	1	**1**	**1**

Students are asked to verify that these equations correctly implement the addition of 1-bit binary numbers (unsigned integers).

With the full adder concept in place, it is straightforward to implement an *n*-bit adder, by cascading *n* full adders, where the carry-out of the current bit serves as the carry-in of the next bit. The equations relating the input variables to output variables follow:

$$\begin{gathered}
z_1 = x_1 \oplus y_1 \oplus c_0; \quad c_1 = (x_1 \wedge y_1) \vee ((x_1 \oplus y_1) \wedge c_0) \\
z_2 = x_2 \oplus y_2 \oplus c_1; \quad c_2 = (x_2 \wedge y_2) \vee ((x_2 \oplus y_2) \wedge c_1) \\
z_3 = x_3 \oplus y_3 \oplus c_2; \quad c_3 = (x_3 \wedge y_3) \vee ((x_3 \oplus y_3) \wedge c_2) \\
\cdots\cdots\cdots\cdots \\
z_{n-1} = x_{n-1} \oplus y_{n-1} \oplus c_{n-2}; \quad c_{n-1} = (x_{n-1} \wedge y_{n-1}) \vee ((x_{n-1} \oplus y_{n-1}) \wedge c_{n-2}) \\
z_n = x_n \oplus y_n \oplus c_{n-1}; \quad c_n = (x_n \wedge y_n) \vee ((x_n \oplus y_n) \wedge c_{n-1})
\end{gathered}$$

The above equations realize the addition of *n*-bit binary numbers *X* and *Y*: $(x_n \ldots x_1)_2 + (y_n \ldots y_1)_2 + c_0 = (c_n z_n \ldots z_1)_2$.

☶

16. (***) Kleene logic

In Table 3.2, we show that when logic connectives AND and OR are used as multiplication and addition operators, respectively, we have a Boolean algebra, the Boolean logic of which is different from the familiar logic of school algebra.

We also have shown that given any Boolean function, there exists a Boolean expression that implements the Boolean function. In other words, any Boolean function can be implemented by AND, OR, NOT operators over Boolean constants and Boolean variables. This beautiful property should not be taken for granted. A slight change could nullify this property. Let us look at an example.

The set of all **Kleene expressions** *L* is recursively defined as follows.

1. Initially, let 0, 1, $x_1, \cdots, x_n \in L$, where 0 and 1 are *Kleene constants*, and $x_1, \cdots, x_n$ are *Kleene variables*.
2. Recursively apply NOT, AND, OR to *L*: If $x, y \in L$, then $\neg x$, $x \bullet y$, $x + y \in L$. Use the basic properties in Table 3.5 (called **Kleene algebra axioms**) to reduce all equivalent expressions into one expression, that is, to eliminate redundancy.
3. Repeat Step 2 until *L* no longer changes.

Note that the Complementation law $x + \neg x = 1$ does not hold anymore. It is replaced by three weaker laws: the de Morgan law, the double negation law, and the product law. This seemingly slight change to Boolean logic makes the following statement to be false: any Kleene function can be implemented by AND, OR, NOT operators over Kleene constants and Kleene variables.

Table 3.5 Contrasting the axioms of Boolean algebra and Kleene algebra

Law	Boolean Algebra	Kleene Algebra
Associativity	(x • y) • z = x • (y • z),	(x • y) • z = x • (y • z),
	$(x + y) + z = x + (y + z)$	$(x + y) + z = x + (y + z)$
Commutativity	$x • y = y • x$	$x • y = y • x$
	$x + y = y + x$	$x + y = y + x$
Distributivity	$(x + y) • z = (x • z) + (y • z)$	$(x + y) • z = (x • z) + (y • z)$
	$(x • y) + z = (x + z) • (y + z)$	$(x • y) + z = (x + z) • (y + z)$
Identity	$x + 0 = x, x • 1 = x$	$x + 0 = x, x • 1 = x$
Annihilator	$x • 0 = 0, x + 1 = 1$	$x • 0 = 0, x + 1 = 1$
Idempotence	$x • x = x, x + x = x$	$x • x = x, x + x = x$
Absorption	$(x•y)+x = x, (x+y)•x = x$	$(x•y)+x = x, (x+y)•x = x$
Complementation	$x + \neg x = 1, x • \neg x = 0$	N/A
de Morgan		$\neg(x+y)= \neg x • \neg y, \neg(x • y)= \neg x + \neg y$
Double Negation		$\neg\neg x = x$
Product		$p•x_i+p•\neg x_i = p+p•x_i+p•\neg x_i$

In Boolean algebra, $p•x_i+p•\neg x_i = p•(x_i+\neg x_i)= p$. But in Kleene algebra, this no longer holds. We have $p•x_i+p•\neg x_i = p+p•x_i+p•\neg x_i$, where p is a product of Kleene variables and their negations. For instance, given the product $p = \neg x_1•x_2$, from the product law we have $(\neg x_1•x_2)•x_3+(\neg x_1•x_2)•\neg x_3 = \neg x_1•x_2 + \neg x_1•x_2•x_3 + \neg x_1•x_2•\neg x_3$, not $(\neg x_1•x_2)•x_3+(\neg x_1•x_2)•\neg x_3 = \neg x_1•x_2$.

We know that there are 2^{2^n} distinct Boolean expressions of n variables, with the assumption that equivalent expressions are counted as one expression.

How many distinct Kleene expressions of n variables are there, for a given n>0? This is still an open problem. We compare the numbers of distinct Boolean and Kleene expressions of n variables in Table 3.6. Note that we still do not know the formula for the number of distinct Kleene expressions, but do know that this number is less than 2^{3^n}.

Table 3.6 The numbers of distinct Boolean and Kleene expressions of n variables

n	# of Distinct Boolean Expressions	# of Distinct Kleene Expressions
1	$2^{2^1} = 4$	6
2	$2^{2^2} = 16$	84
3	$2^{2^3} = 256$	43918
4	$2^{2^4} = 65536$	160297985276
In general	2^{2^n}	Unknown but < 2^{3^n}

To have a more concrete understanding, let us enumerate all Kleene expressions of one variable *x*. Initially, the set *L* of Kleene expressions is $L = \{0,1,x\}$.

Round 1. We obtain a new expression $\overline{x}$. The new $L = \{0, 1,\ x;\ \overline{x}\}$.

Round 2. For Boolean expressions, we would stop here, as *L* no longer changes.

Round 2. For Kleene expressions, we continue and obtain two new expressions $x \cdot \overline{x}, x + \overline{x}$, the new $L = \{0, 1,\ x;\ \overline{x};\ x \cdot \overline{x}, x + \overline{x}\}$.

Round 3. Stop, since *L* no longer changes.

Note that the number of Boolean expressions of one variable *x* is 2^{2^1}=4. These four expressions are: always false, always true, identical to *x*, and NOT *x*. For Kleene logic, the number of Kleene expressions is 6. The two new expressions are: *x* AND (NOT *x*), and *x* OR (NOT *x*), which are absent from Boolean logic.

We also have the notion of Kleene functions, similar to Boolean functions. An *n*-input-1-output **Kleene function** is a mathematical function

$$f: \{0, \mathrm{I}, 1\}^n \rightarrow \{0, \mathrm{I}, 1\},$$

and the mapping is defined by the truth table of *f*. Different from binary Boolean logic, Kleene logic is ternary, in that we have three values. Besides 0 (False) and 1 (True), we have a new middle value I for Indeterminate. Table 3.7 shows the truth values of the AND, OR, NOT operators on two variables *x* and *y*.

Table 3.7 Truth table showing the truth values of AND, OR, NOT operators in Kleene logic

x	y	$x \wedge y$	$x \vee y$	$\neg x$
0	0	**0**	**0**	**1**
0	I	**0**	**I**	**1**
0	1	**0**	**1**	**1**
I	0	**0**	**I**	**I**
I	I	**I**	**I**	**I**
I	1	**I**	**1**	**I**
1	0	**0**	**1**	**0**
1	I	**I**	**1**	**0**
1	1	**1**	**1**	**0**

Note that there are 3^{3^n} *n*-input-1-output distinct Kleene functions and truth tables. For *n*=1, there are $3^{3^1} = 27$ distinct Kleene functions but only 6 distinct Kleene expressions. Many Kleene functions cannot be represented by Kleene expressions. Thus, in Kleene logic, some Kleene functions cannot be implemented by AND, OR, NOT.

17. Using propositional logic to solve problems

We discuss four examples to show how logic helps produce correct computational processes.

Example 17. The congruent triangles problem, revisited

Given any conditional statement "if P then Q", we denote it in propositional logic as P→Q. The negation of this statement is NOT(if P then Q), or ¬(P→Q). The two statements are related. Only one is true.

Using the triangles example, let us call "P→Q" the original statement, where P stands for "two triangles are congruent" and Q stands for "two triangles are similar". Four types of statements are derived from the original statement, as shown in Table 3.8. These four types of statements are logically related.

Table 3.8 The Converse, Inverse, Contrapositive, and Negation of a conditional statement

Statement Type	Logic Form	Triangles Example
Original	P→Q	If two triangles are congruent, then they are similar.
Converse	Q→P	If two triangles are similar, then they are congruent.
Inverse	(¬P)→(¬Q)	If two triangles are not congruent, then they are not similar.
Contrapositive	(¬Q)→(¬P)	If two triangles are not similar, then they are not congruent.
Negation	¬(P→Q)	NOT (If two triangles are congruent, then they are similar.)

These logic relationships can be used to arrive at new statements and their truth values, sometimes without needing domain knowledge of geometry.

The original statement, "if two triangles are congruent, then they are similar", is a true statement. We know this from geometry. Congruent triangles have the same shape and size. Similar triangles have the same shape. Two triangles, having the same shape and size, of course have the same shape. Thus, they are similar. That is, P→Q is a true proposition.

The **converse** of the original statement is "if two triangles are similar, then they are congruent". The converse of the conditional statement P→Q is the statement obtained by exchanging the position of P and Q, namely, Q→P. From geometry knowledge, we know this statement is false. Two similar triangles can have the same shape but different sizes, thus are not congruent. That is, Q→P is a false proposition.

The **inverse** of the original statement is "if two triangles are not congruent, then they are not similar". The inverse of the conditional statement P→Q is the statement obtained by negating both P and Q, namely, (¬P)→(¬Q).

Now, what is the truth value of the inverse statement? Is it true or false? We can obtain the answer without knowing geometry. In fact, we know immediately that the inverse statement (¬P)→(¬Q) in this example is a false proposition, because (1) the converse statement is false, and (2) *the converse and the inverse statements*

are logically equivalent. The second point can be proven easily by showing that $(\neg P)\rightarrow(\neg Q) = Q\rightarrow P$.

$(\neg P)\rightarrow(\neg Q)$	The given inverse statement
$= \neg(\neg P) \vee (\neg Q)$	by implication property $P\rightarrow Q = \neg P \vee Q$
$= P \vee (\neg Q)$	eliminate double negation
$= \neg Q \vee P$	use communicative law
$= Q\rightarrow P$	obtain the converse by implication property.

The **contrapositive** of the original statement is "if two triangles are not similar, then they are not congruent". The contrapositive of the conditional statement $P\rightarrow Q$ is the statement obtained by negating both P and Q, and then exchanging positions, namely, $(\neg Q)\rightarrow(\neg P)$. We can show that *the contrapositive statement and the original statement are logically equivalent.* That is, $(\neg Q)\rightarrow(\neg P)$ is equivalent to $P\rightarrow Q$. The proof is the same as showing $(\neg P)\rightarrow(\neg Q) = P\rightarrow Q$.

Finally, the **negation** of the original statement is "NOT (If two triangles are congruent, then they are similar)". The negation of the conditional statement $P\rightarrow Q$ is just the logic negation, namely, $\neg(P\rightarrow Q)$. It can be transformed into other equivalent forms: $\neg(P\rightarrow Q) = \neg(\neg P \vee Q) = (\neg\neg P) \wedge (\neg Q) = P \wedge (\neg Q)$. That is, "Two triangles are congruent AND they are not similar", which is a false statement.

Note that the negation is not the same as the inverse. The original statement AND its negation always form a contradiction: $(P\rightarrow Q) \wedge \neg(P\rightarrow Q) = \text{FALSE}$. The original statement AND its inverse yield "P is identical to Q": $(P\rightarrow Q) \wedge (\neg P\rightarrow\neg Q) = P\equiv Q$.

☲

Example 18. The impatient guide problem, revisited

A tourist is traveling in the land of Oz and wants to go to the Emerald City. The tourist reaches a crossroad with paths P and Q, one of which leads to the Emerald City. There is a guide G at the crossroad, who comes from either the Honest Village or the Lying Village. Anyone from the Honest Village always tells the truth, and anyone from the Lying Village always tells lies. The guide is impatient, in that G only answers one question from the tourist, and the answer is either "Yes" or "No".

What question should the tourist ask the guide, to determine the correct path?

The main difficulty is as follows. On one hand, the tourist needs to collect information which apparently needs at least two answers to Yes-No questions. On the other hand, the impatient guide only answers one question. Fortunately, propositional logic tells us that we can use proper connectives to combine two propositions into a single proposition. One of the propositions should contain information about the Honest or Lying Village, the other should be about the path to the Emerald City. With this line of thought, we come up with the following question:

Are your answers the same, to the two questions "are you from the Honest Village" and "does path P lead to the Emerald City"?

If the answer is "Yes", take path P; if the answer is "No", take path Q.

To make the above reasoning clearer, let us use propositional notations to denote the solution.

- H denotes the proposition "G is from the Honest Village". That is, H=1 means G is from the Honest Village; H=0 means G is from the Lying Village.
- S denotes the proposition "Path P leads to the Emerald City". That is, S=1 means path P leads to the Emerald City; S=0 means path Q leads to the Emerald City.

With these notations, the single question to ask is ¬(H⊕S)=?. However, the answer we get is not the true value of ¬(H⊕S) since it depends on whether the guide G comes from Honest Village or not. If G is from the Honest Village, we will get the true value of ¬(H⊕S); while if G is from the Lying Village, we will get the true value of H⊕S. If we use the propositional notations to represent the above argument, the answer we will hear is actually (¬H)⊕ ¬(H⊕S). By applying the properties in Table 3.2 and 3.3 or calculating the truth table of this Boolean expression, it is easy to find that (¬H)⊕ ¬(H⊕S) = S. Thus, we should choose path P if the answer we get is "Yes" and choose path Q if the answer we get is "No".

To make the argument clearer, we verify the correctness of the question and its answer by using a truth table.

H	S	¬(H⊕S)	Comments
0	0	1	G is lying and the true value of the question is "Yes". The answer is "No", take path Q.
0	1	0	G is lying and the true value of the question is "No". The answer is "Yes", take path P.
1	0	0	G is telling the truth and the true value of the question is "No". The answer is "No", take path Q.
1	1	1	G is telling the truth and the true value of the question is "Yes". The answer is "Yes", take path P.

☲

Example 19. The parity program to show logic and bit-shift operations

The **parity** of a number refers to whether the number's bits have an even number of 1's (parity is 0) or an odd number of 1's (parity is 1). The program parity.go below computes the parity values of 63 and 127. The parity function computes parityValue = $X_0 \oplus X_1 \oplus X_2 \oplus \ldots \oplus X_{63}$ for any 64-bit integer $X = (X_{63}X_{62} \ldots X_0)_2$, where ⊕ is the XOR operator. So, 63 = 0…00111111 has six 1's (even, parity is 0), and 127 = 0…01111111 has seven 1's (odd, parity is 1).

The statement

parityValue ^= X & 1

is a shorthand for

parityValue = parityValue ^ (X & 1)

where & is the bitwise AND operator and ^ is the bitwise XOR operator. More specifically, let $X = (X_{63}X_{62} \ldots X_0)_2$ and $Y = (Y_{63}X_{62} \ldots Y_0)_2$, we have $X \,\&\, Y = (X_{63} \wedge Y_{63}, X_{62} \wedge Y_{62}, \ldots X_0 \wedge Y_0)_2$ and $X \,\hat{}\, Y = (X_{63} \oplus Y_{63}, X_{62} \oplus Y_{62}, \ldots X_0 \oplus Y_0)_2$.The expression (X & 1) clears all bits of X but keeps the rightmost bit intact, that is, $X \,\&\, 1 = (00 \ldots 00X_0)_2$. The expression parityValue ^ (X & 1) computes (the current parityValue) $\oplus$ (last bit of X). Finally, the statement X = X >> 1 right shifts X one bit, for the next iteration. That is, X >> 1= $(0X_{63}X_{62} \ldots X_2X_1)$.

```
package main
import "fmt"
func parity(X int) int {
  parityValue := 0
  for i := 0; i<64; i++ {          // X is a 64-bit integer
    parityValue ^= X & 1           // parityValue = parityValue ^ (X & 1)
    X = X >> 1                     // shift X right one bit
  }
  return parityValue
}
func main() {
  a := 63                          // 63 = 00111111 has six 1's
  fmt.Println(parity(a))
  a = 127                          // 127 = 01111111 has seven 1's
  fmt.Println(parity(a))
}
```

(a) Source code of program parity.go

```
> go run parity.go
0
1
>
```

(b) Running parity.go to produce the output

Fig. 3.1 Using parity.go to illustrate logic and shift operations

Example 20. Program to hide a character in a byte array

In the Text Hider project, students are asked to hide a text file in an image file. This example does a much simpler task of hiding an ASCII character 'K' in a byte array A=[11010001, 11001001, 11011010, 11011010] = [D1, C9, DA, DA]. The program replace.go in Fig. 3.7 does this by replacing the least significant two bits of the four elements of array A, with the eight bits of character 'K'. Every element A[i] hides two bits of 'K', as the following table shows.

Table 3.9 Values of elements of array A before and after replacing the least significant two bits with character 'K' = 75 = 01001011.

Array Element	Before	After
A[0]	11010001	11010011
A[1]	11001001	11001010
A[2]	11011010	11011000
A[3]	11011010	11011001

```
package main
import "fmt"
func main() {
   A := [4]byte{0xD1,0xC9,0xDA,0xDA}
   fmt.Printf("Before: \tA = [%b %b %b %b]\n",A[0],A[1],A[2],A[3])
   data := byte('K')
   for i := 0; i < len(A); i++ {
      v := data & 0x3                  // retain last 2 bits of 'K'
      A[i] = A[i] & 0xFC               // clear last 2 bits of A[i]
      A[i] = A[i] | v                  // set last 2 bits of A[i] with those of 'K'
      data = data >> 2                 // repeat with the next 2 bits of 'K'
   }
   fmt.Printf("After: \t\tA = [%b %b %b %b]\n",A[0],A[1],A[2],A[3])
}
```

(a) Source code of program replace.go

```
> go run replace.go
Before:          A = [11010001 11001001 11011010 11011010]
After:           A = [11010011 11001010 11011000 11011001]
>
```

(b) Running replace.go to produce the output

Fig. 3.2 Using replace.go to illustrate logic and shift operations

The program uses tab \t to align the two lines of printing outputs, to better see the changes made to the last two bits of the array elements. The main work is done in the for loop, which iterates over the four elements A[0] to A[3], with the index values changing from i=0, 1, 2, to 3. We only need to look at the detailed case when i=0. The other cases are similar. When i=0, we have data='K'=01001011 and A[i]=A[0]= 11010001. The results of the loop body are shown below step-by-step.

v := data & 0x3	v=	01001011 & 00000011 = 00000011
		// retain rightmost 2 bits of 'K'
A[i] = A[i] & 0xFC	A[i]=	11010001 & 11111100 = 11010000
		// clear rightmost 2 bits of A[i]
A[i] = A[i] | v	A[i]=	11010000 | 00000011 = 11010011
		// set last 2 bits of A[i] with those of 'K'
data = data >> 2	data=	00010010
		// shift 'K' 2 bits to the right

☶

3.1.2. Predicative Logic

Predicative logic is also called first-order logic. It contains propositional logic as well as predicates and quantifiers. Predicative logic has more expressive power than propositional logic. That is, predicative logic can be used to rigorously express some logic statements which propositional logic cannot do.

18. Predicate and quantifier

A **predicate** can be viewed as a proposition with one or more input variables. A predicate takes an entity or entities as input variables to produce an output of either True or False value. For instance, consider the two sentences "Socrates is a philosopher" and "Plato is a philosopher". In propositional logic, these sentences are viewed as being unrelated and may be denoted, for example, by propositional variables p and q. However, in predicative logic, we can use predicate Phil(x) to represent "x is a philosopher", where x is an input variable. Thus, if "a" represents "Socrates", then Phil(a) means "Socrates is a philosopher". Phil(x) is a predicate with one input variable x. Phil(a) is a predicate with the input variable x instantiated with the entity constant a.

There are two **quantifiers** in predicative logic. The universal quantifier ∀ means "for all", "for any", "for every". The existential quantifier ∃ means "there exists". For instance, we can have the following statements and their expressions.

All philosophers are mortals.	$\forall x\ [\text{Phil}(x) \rightarrow \text{Mortal}(x)]$.
Socrates is a philosopher.	Phil(a).
Socrates is a mortal.	Mortal(a).
There exists a philosopher.	$\exists x\ [\text{Phil}(x)]$.

In predicative logic, we need to pay attention to the quantifies about their domain, order and use with negation, as illustrated by the following example.

Example 21. Domain, order, and negation when using quantifiers

Consider the mathematical statement: for any natural number n, either n is an even number, or n+1 is an even number. This statement expressed in natural language can be more concisely and precisely expressed in predicative logic as

$$\forall n\ [\text{Even}(n) \vee \text{Even}(n+1)],$$

where we use predicate $\text{Even}(n)$ to mean n is an even number.

However, the above predicative logic expression does not consider "for any natural number". This can be compensated by explicitly indicating the **domain** of variable *n* associated with the universal quantifier, and the expression becomes:

$$\forall n \in \mathbb{N}\ [\text{Even}(n) \vee \text{Even}(n+1)],$$

where $\mathbb{N}$ represents the set of natural numbers.

In predicative logic, the **order** of the quantifiers is important. Look at the following two statements. The first is true while the second is false.

(1) $\forall\, x \in \mathbb{N}, \exists\, y \in \mathbb{N}\ (y = x + 1)$ Every nutural number has a successor.

(2) $\exists\, y \in \mathbb{N}, \forall\, x \in \mathbb{N}\ (y = x + 1)$ There is a nutural number which is the successor of all nutural numbers.

With negation, we need to differentiate "Not-All" and "All-Not" statements. More precisely, consider the following two statements.

$\neg(\forall\, x \in \mathbb{N}\ [\text{Even}(n)])$ Not all natural numbers are even.

$\forall\, x \in \mathbb{N}\ [\neg\text{Even}(n)])$ All natural numbers are not even.

The first statement (Not-All) happens to be true, and the second statement (All-Not) happens to be false.

Negation with quantifiers satisfies the following **negation properties**:

$$\neg\big(\exists x\ \text{P}(x)\big) = \forall x\ \neg\text{P}(x),\ \ \neg\big(\forall x\ \text{P}(x)\big) = \exists x\ \neg\text{P}(x).$$

For instance, the true statement

$\neg(\forall\, x \in \mathbb{N}\ [\text{Even}(n)])$ Not all natural numbers are even

is equivalent to

$\exists\, x \in \mathbb{N}\ [\neg\text{Even}(n)]$ There exists a natural number that is not even.

The false statement

$\neg(\exists x \in \mathbb{N}\ [\text{Even}(n)])$ There exists no natural number that is even

is equivalent to

$\forall\, x \in \mathbb{N}\ [\neg\text{Even}(n)]$ All natural numbers are not even.

The negation properties can be used in cascade. For instance, the statement

$\neg(\exists\, y \in \mathbb{N}, \forall\, x \in \mathbb{N}\ (y \neq x + 1))$ There is no natural number which is not the successor of any natural number

is equivalent to

$\forall y \in \mathbb{N}, \exists\, x \in \mathbb{N}\ (y = x + 1)$ Any natural number is the successor of some natural number.

This statement is false, as zero is not the successor of any natural number.

☶

19. More examples of writing predicative logic expressions

We discuss more examples to show how to write predicate logic expressions. In particular, we show how natural language statements can be expressed as predicate logic expressions. The latter can express the statements more rigorously.

Example 22. Representing infinity

Consider the following statement expressed in natural language:

> There exist infinitely many prime numbers.

How to represent this statement as a predicate logic expression?

We use the predicate Prime(m) to represent "m is a prime number". Now the key is how to express "infinite". There are several ways to do it.

The first way is to express "infinite" directly by interpreting it. We convert the original statement "there exist infinitely many prime numbers" into a more concrete statement: "for any natural number, these exists some prime number larger than it". Since there are infinitely many natural numbers, there are infinitely many prime numbers.

Then, "there exist infinitely many prime numbers" can be expressed by the following expression:

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, [(m > n) \wedge (\text{Prime}(m))]$$

which is a direct rewriting of the more concrete statement.

The second way is to is to express "infinite" as "not finite". We first write a statement for "finite", and then negate it. The following two expressions are examples of this method. We use the idea: for any finite subset of $\mathbb{N}$, there exists a maximum number in this subset.

$$\neg(\exists n \in \mathbb{N}, \forall m \in \mathbb{N}, \left[(m > n) \rightarrow \neg\big(\text{Prime}(m)\big)\right])$$

$$\neg(\exists n \in \mathbb{N}, \forall m \in \mathbb{N}, [(\text{Prime}(m)) \rightarrow (m \leq n)])$$

We leave it as an exercise to show that the above two expressions indeed express the statement "there exist infinitely many prime numbers".

☶

Example 23. Predicate refinement

We use the predicate Prime(m) to represent "m is a prime number" in the above example, to obtain the following expression for "there exist infinitely many prime numbers":

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, [(m > n) \wedge (\text{Prime}(m))]$$

Let us try to express "prime number", by refining the predicate Prime(m). From mathematics, prime numbers are positive integers which have only 2 factors: 1 and themselves. This definition of prime number is a little bit difficult to express, because it contains the phrase "have only". We can use an equivalent but easier-to-express definition: a prime number is a natural number that cannot be generated by multiplying other two natural numbers greater than 1. Thus, we have

$$\text{Prime}(m) = \forall \text{p}, \text{q} \in \mathbb{N}, \text{p}, \text{q} > 1(m \neq pq).$$

Here, $\text{p}, \text{q} \in \mathbb{N}, \text{p}, \text{q} > 1$ is the domain of variable p, q. Intuitively, the above equation says that *m* is a prime number if *m* is not the product of two natural numbers *p* and *q* which are both greater than 1.

We can directly substitute $\text{Prime}(m)$ in the original expression:

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, \big[(m > n) \wedge \big(\forall p, q \in \mathbb{N}, p, q > 1(m \neq pq)\big)\big].$$

But usually, we write all quantifiers in front of the expression. Thus, we have:

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, \forall p, q \in \mathbb{N}, p, q > 1[(m > n) \wedge (m \neq pq)].$$

Here, we emphasize again the importance of the order of the quantifiers. Consider the following two expressions

$$\forall n \in \mathbb{N}, \forall p, q \in \mathbb{N}, p, q > 1, \exists m \in \mathbb{N}\ [(m > n) \wedge (m \neq pq)]$$

and

$$\exists m \in \mathbb{N}, \forall n \in \mathbb{N}, \forall p, q \in \mathbb{N}, p, q > 1[(m > n) \wedge (m \neq pq)].$$

Neither expression is equivalent to the statement "there exist infinitely many prime numbers".

Now consider another related logic statement called the twin prime conjecture:

> There are an infinite number of twin prime pairs.

How to represent this statement as a predicate logic expression? The key here is what is "twin prime pair". Twin primes (or a twin prime pair) are two prime numbers with a difference of 2. For instance, 5 and 7 form a twin prime pair, so do 11 and 13. Adding this definition to the original expression for infinite prime numbers, we have a predicate logic expression for the twin prime conjecture as follows:

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, \forall p, q \in \mathbb{N}, p, q > 1[(m > n) \wedge (m \neq pq) \wedge (m + 2 \neq pq)].$$

☶

Example 24. Representing potentially unbounded process

Let us consider the following statement called the Collatz conjecture, which is a logic statement involving a potentially unbounded process.

> For any positive integer *n*, multiply *n* by 3 and add 1 if *n* is odd, and divide *n* by 2 if *n* is even. Repeat this process and you will always get 1.

For example, for $n = 15$, the above process become $15 \rightarrow 46 \rightarrow 23 \rightarrow 70 \rightarrow 35 \rightarrow 106 \rightarrow 53 \rightarrow 170 \rightarrow 85 \rightarrow 256 \rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$. After 17 steps, the process converges to 1.

We use $f(n)$ to represent one step for integer *n*, and use $f^{(m)}(n)$ to represent the m times composition of function f, that is, $f\ (f\ (\cdots\ f\ (n)\ \cdots))$. The Collatz conjecture can be expressed by the following predicate logic expression:

$$\forall n, \exists m, [f^{(m)}(n) = 1], \text{ where } f(n) = \begin{cases} 3n + 1, & \text{if } n \equiv 1 \pmod 2; \\ n/2, & \text{if } n \equiv 0 \pmod 2. \end{cases}$$

At present, this conjecture has not yet been solved.

☶

20. Inference rules and axiomatic systems in Boolean logic

We have implicitly used **axioms** and **inference rules** from school logic in understanding the material of Boolean logic. On the other hand, we also point out that Boolean algebra is different from school algebra. This seemingly contradiction raises a question: is logic thinking different in computer science from that in ordinary mathematics? How to rigorously specify the difference?

Normally, logic thinking in computer science is the same as that in ordinary mathematics. More specifically, we can use a mathematic method called **axiomatic systems** to specify any particular logic system. An axiomatic system is built from three components: (1) a set of *elements and operators* on these elements, (2) a set of *axioms*, i.e., given properties about the elements and operators; and (3) a set of *inference rules* to derive new properties from known properties.

To rigorously specify a logic system in computer science, such as Boolean logic, that is different from ordinary school mathematics, we explicitly specify different operators and axioms but normally **use the same inference rules of mathematics**. For instance, comparing to algebra in high school mathematics, Boolean logic introduces a new NOT operator. In addition, as shown in Table 3.2, Boolean logic introduces three new axioms (the idempotence, the absorption, and the complementation laws) and changes the distributivity and the annihilator laws.

We explicitly list three sets of commonly used inference rules in Box. 8. They are all inference rules of ordinary mathematics, and can be used to infer a statement (conclusion), given one or more statements (premises).

Box 8. Several Commonly Used Inference Rules

Modus Ponens:

Given	$X \rightarrow Y$	Every Web page has a URL
	$\underline{X\quad}$	My homepage is a Web page
Conclude	Y	My homepage has a URL

Modus Tollens:

Given	$X \rightarrow Y$	Every Web page has a URL
	$\underline{\neg Y\quad}$	My cellphone does not have a URL
Conclude	$\neg X$	My cellphone is not a Web page

Negating Quantified Predicate:

Given	$\neg(\exists x\, P(x))$	Given	$\forall x\, \neg P(x)$
Conclude	$\forall x\, \neg P(x)$	Conclude	$\neg(\exists x\, P(x))$
Given	$\neg(\forall x\, P(x))$	Given	$\exists x\, \neg P(x)$
Conclude	$\exists x\, \neg P(x)$	Conclude	$\neg(\forall x\, P(x))$

3.2. Automata and Turing Machines

When a computational process has a single step, Boolean logic often suffices to produce correct answer and ensures logic correctness. However, when a computational process involves multiple steps, we often prefer new models. A key concept is *automata*, also known as state machines. An automaton can remember things by holding states and use state transitions to represent steps.

David Hilbert (1862-1943) put forward a very fundamental and general problem that requires multi-step computational processes, the *Entscheidungsproblem* (the decision problem), i.e., mechanically proving theorems of mathematics. Alan Turing gave a negative answer to the decision problem, but in the process, proposed Turing machines, a class of automata that turn out to be able to solve any computable problems. Turing machines are key milestones and cornerstones of correctness and generality of computational processes.

Fundamental limitations of computation are also discussed. There exist incomputable problems that cannot be solved by any Turing machine. We also have Gödel's incompleteness theorem: being true and being provable are not the same thing. In any reasonably sophisticated mathematic system, there are mathematic theorems which cannot be proven.

3.2.1. Mechanical Theorem Proving

Mechanical theorem proving (also known as automated theorem proving or computer-assisted proof), requires that in the process of calculation or proof, after each step, there is a certain rule to choose the next step. Along this path, the process will finally reach the required conclusion. In this way, people hope to avoid those highly skilled mathematical calculations or proofs and replace them with the powerful computing power of modern computers.

The idea of mechanical proof can be traced back to the 17th century French mathematician Rene Descartes (1596-1650). Descartes once had a great idea: "All problems can be turned into mathematical problems; all mathematical problems can be turned into algebraic problems; and all algebraic problems can be turned into solving algebraic equations." Descartes created analytical geometry, established the bridge between the spatial form and the quantitative relationship, and established the framework to solve elementary geometric problems based on algebraic methods.

In 1928, David Hilbert stated the problem of mechanical theorem proving more clearly: given an axiomatic system, is there a mechanical method (now called an algorithm) that can verify the truth or falsity for every proposition in this system? In Section 3.2.3 we will see that the answer to this Entscheidungsproblem is No.

However, although it is impossible to use an algorithm to determine all the propositions, it is still feasible to use mechanized methods for specific problems in specific fields. For example, the elimination method (Wu's method) based on the zero-point set of the polynomial system proposed by Professor Wenjun Wu (also known as Wu Wen-tsün,) can be applied to the mechanical proof of a large number of geometric theorems.

The first major theorem proved with the help of computer is the four-color theorem. This famous four-color theorem in graph theory asserts that any planar graph can be 4-colored, that is, there is a way to dye each vertex with one of four colors so that any adjacent vertices do not have the same color. The four-color theorem was first proposed by Francis Guthrie (1831-1899) in 1852. This problem puzzled mathematicians for more than a hundred years. It was finally proved by Kenneth Appel and Wolfgang Haken in 1976 with the help of computer.

The idea of their proof is as follows: if a certain structure appears in the planar graph, this part can be replaced with a smaller structure (that is, reduce the size of the original graph), while the 4-colored property is unchanged. That is, if the new graph can be 4-colored, the original larger graph can also be 4-colored. For example, a vertex with a degree no greater than 3 can be removed, because it does not affect whether the whole graph can be colored by 4 colors. Appel and Haken proved that there are 1936 planar graphs that cannot be reduced to one another. Any other planar graph can always reach one of these 1936 graphs through the specific process of reduction. Finally, with the help of the computer, after more than 1,000 hours of calculation, they verified that all 1936 graphs can be 4-colored and thus proved the four-color theorem.

3.2.2. Automata

When a proof process is viewed as a computational process, it is usually a multistep process. We start from the axioms or a known true statement (a theorem), and repetitively apply the inference rules to arrive at new true statements and the final conclusion. The result of an inference step should be memorized as a state, and used as part of inputs for future steps. That is, we need a machine that holds states. Such a machine is called an **automaton**. We will introduce two classes of automata, namely finite state automata and Turing machines.

Consider a simple vending machine, which sells bottled water and bagged biscuits. The price of bottled water is $1 per bottle and the price of biscuits is $2 per bag. The vending machine accepts only $1 or $2 banknotes. In any state, a buyer can perform one of five actions to the vending machine: (1) insert a $1 banknote, (2) insert a $2 banknote, (3) press the "Buy water" button, and (4) press the "Buy biscuits" button, and (5) press the "Get money back" button.

Initially, the vending machine is in the state q_0 (initial state). If the buyer inserts a $1 banknote, the vending machine will transfer to a new state q_1. If the buyer chooses to buy bottled water in state q_1, the vending machine will output one bottle of water and go back to state q_0. If the buyer inserts one more $1 banknote in state q_1, the vending machine will transfer to a new state q_2. If the buyer chooses to buy something in state q_2, the vending machine will output the corresponding goods and go back to q_0 (biscuits) or q_1 (bottled water). If the buyer wants to get the money back in state q_1 or q_2, the vending machine will return the corresponding amount of money and go back to state q_0.

Figure 3.3 is called the **state-transition diagram**, which shows the above state transition rules of the vending machine. Note that the arrow notation specifies an

input-output pair. "$1→$1"at the arrowed curve in state q_2 denotes "when the buyer inserts a $1 bill, the machine outputs a $1 bill and stays in state q_2." The notation "$1→" at the arrowed curve from state q_0 to state q_1 denotes " when the buyer inserts a $1 bill, the machine outputs nothing and transitions from state q_0 to state q_1."

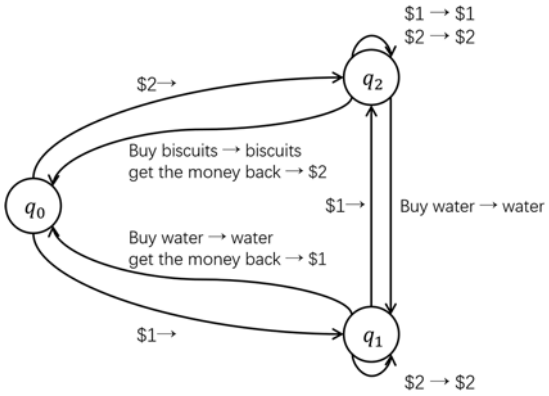


Fig. 3.3 The state transition diagram for a vending machine

The state-transition diagram can be equivalently written as a **state-transition table** in Table 3.10. Note that the state transition diagram happens to omit some possible transitions, while the state transition table lists all possible transitions.

Table 3.10 State transition table of a vending machine

Current State	Input	Output	Next State
q_0	Insert $1	Null	q_1
	Insert $2	Null	q_2
	Buy water	Null	q_0
	Buy biscuits	Null	q_0
	Get money back	Null	q_0
q_1	Insert $1	Null	q_2
	Insert $2	Output $2	q_1
	Buy water	Output water	q_0
	Buy biscuits	Null	q_1
	Get money back	Output $1	q_0
q_2	Insert $1	Output $1	q_2
	Insert $2	Output $2	q_2
	Buy water	Output water	q_1
	Buy biscuits	Output biscuits	q_0
	Get money back	Output $2	q_0

The above computational model for the vending machine is called a (deterministic) **finite automaton**, also known as a finite-state automaton. It is a model of computation suitable for computational processes where only finite numbers of states are involved. A computational process that involves potentially infinite number of states cannot be modeled by a finite-state automaton. Finite automata cannot handle infinite states, as shown by the following example.

Example 25. Palindromes cannot be recognized by finite automata

A palindrome is a character string that is the same when reading backwards. For instance, 1991 is a palindrome, so is 0100110001110000111100001110001100 10. We leave it as an exercise for students to show that palindromes cannot be recognized by finite automata.

☶

3.2.3. Computation on Turing Machine

What is computable? Alan Turing gave a rigorous definition in his famous paper in 1936. His idea is that all the infinite mathematical entities, such as numbers, variables, functions and predicates, can be mapped to the infinite set of real numbers. A mathematical entity is computable if its corresponding real number is computable. What is computable is reduced to what real numbers are computable. "The computable numbers [are] the real numbers whose expressions as a decimal are calculable by finite means." Another equivalent definition is: "A number is computable if its decimal can be written down by a machine."

Example 26. The circular constant π is computable

We use π to denote the circular constant (the ratio of circumference to diameter of any circle). It is an irrational number and has infinitely many decimal digits. Nevertheless, π is computable according to Turing's definition: π is a real number whose decimal digits are calculable by finite machines. That is, any sequence of digits of π that we want can be produced by finite means.

Suppose we want the first 800 digits of π. This sequence can be produced by the following finite means: running the following pi.go program[2] on a laptop computer. The program is finite, as it contains 27 lines of code. The computer is finite with 2GB memory capacity. The running time is finite, as executing the pi.go program consumes less than 1 second. The entire execution process is automatic.

---

[2] This pi.go program is rewritten into Go code from a 160-character C program written by Dik T. Winter of the Centrum Wiskunde & Informatica (CWI) in the Netherlands. Dr. Ben Lynn of Stanford University analyzed the C code. Please see his analysis note at https://crypto.stanford.edu/pbc/notes/pi/code.html.

```
package main
import "fmt"
func main() {
   var r [2801]int
   var i, k, b, d int
   c := 0
   for i = 0; i < 2800; i++ {
      r[i] = 2000
   }
   for k = 2800; k > 0; k -= 14 {
      d = 0
      i = k
      for ;; {
         d += r[i] * 10000
         b = 2 * i - 1
         r[i] = d % b
         d /= b
         i--
         if i == 0 {break}
         d *= i
      }
      fmt.Printf("%.4d", c + d / 10000)
      c = d % 10000
   }
}
```

The program pi.go produces the first 800 decimal digits of π:

```
31415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446
09556423058223172535940812848111745028410270193852110555964462294895493038841519644288109756659334461284756482337867831652712019091456485669234687580348610454326648213393607260249141273724587006606315588174881520923797096282925409171536436789259036001133053054882046652138414695194151706416094330572703657595919530921861173819326117931051185480744623799624232749567351885752724891227938183011949129833673362440656643086021390373494639522473719070217986094370277053921717629317675238467481846766091940513200056812714526356082778577134275778960917363717872146844090612249534301465495853710507922796892589235420199561121290219608640344181598136297747713099605187072113499999983729780499510597317328160963185
```

☷

In his 1936 paper "On Computable Numbers, with an Application to the Entscheidungsproblem", Alan Turing described an abstract computer which was later called a Turing machine. Turing machines are a more powerful model of computation than finite automata.

The organization of a Turing machine is shown in Fig. 3.4. At a minimum, a Turing machine is comprised of three components: (1) an infinite tape, (2) a read/write head, and (3) a finite state-transition diagram in a finite state controller.

The tape has infinitely many squares (cells) extending to both directions. Each square contains a symbol, such as 0, 1 and blank. The blank symbol can be written explicitly as B to avoid confusion.

Initially, the tape contains the input string between two blanks. All other squares are blank. The head points to the first input symbol (or to the blank square left of the first input symbol). The state-transition table resides in the finite state controller.

The state-transition diagram or the equivalent state transition table governs the behavior of a Turing machine, as illustrated in Fig. 3.4 and Table 3.11. This particular Turing machine does a cleanup. It scans the input string from left to right, erases each 0 or 1 (replacing 0 or 1 by blank B), and stops when reads a @.

The machine starts at initial state q_0 and stops at final state q_1. Sometimes we explicitly name the final state q_1 as Halt, when there is just one final state. At each step, the head reads the symbol in the pointed square, writes an appropriate symbol, and moves the head to left or right, and then the machine transition to the next state.

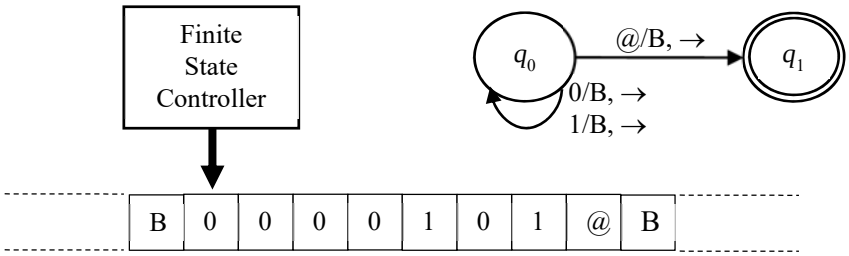


Fig. 3.4 Organization of a Turing machine, with a cleanup function example

Table 3.11 State transition table of a Turing machine

Current State	Symbol Read	Symbol to Write	Head Move	Next State
q_0	0	B	→	q_0
q_0	1	B	→	q_0
q_0	@	B	→	q_1 (Halt)

Definition: A **Turing machine** is a 7-tuple $M = \{Q, \Sigma, \Gamma, \delta, q_0, q_{\text{Accept}}, q_{\text{Reject}}\}$.

- Q is a finite, non-empty set of states.
- Σ is a finite, non-empty set of input symbols.
- Γ is a finite, non-empty set of tape symbols. There is a special character $B \in \Gamma$ for the blank symbol. We require $B \notin \Sigma$ and $\Sigma \subset \Gamma$.
- $\delta: (Q - \{q_{\text{Accept}}, q_{\text{Reject}}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{\rightarrow, \leftarrow\}$ is the transition function.
- $q_0 \in Q$ is the initial state.
- $q_{\text{Accept}} \in Q$ is the accept state.
- $q_{\text{Reject}} \in Q$ is the reject state.

To make the definition concrete, let us review again the cleanup Turing machine illustrated in Fig 3.4 and Table 3.11. For this machine, $Q = \{q_0, q_1\}$, $\Sigma = \{0,1,@\}$, $\Gamma = \{0,1,@,B\}$. The initial state is q_0. The accept state is q_1 which means the machine successfully finishes the cleanup process. There is no reject state. The transition function δ is illustrated in Fig 3.4 and Table 3.11.

If we look at the transition function more carefully, we may find that there is no definition of $\delta(q_0, B)$. When the computational task to erase the input string ends with the symbol @, it is impossible to read B in state q_0. However, it is always useful to write down the rule for all cases in order to handle exceptional conditions. One possible way to handle it is to set $\delta(q_0, B) = (q_2, B, \rightarrow)$ where q_2 is the reject state. This means if the end of the input is not @, the Turing machine will go to the reject state q_2 and stop.

Example 27. Palindromes can be recognized by a Turing machine

In Example 25, we claim that Palindromes cannot be recognized by finite automata. Here, we show it can be recognized by a Turing machine. Thus, Turing machines are more powerful than finite automata.

The Turing machine starts at state q_0 with the input string on the tape, enclosed between two blank squares. The machine has two final states. When the input string is not a palindrome, the machine eventually stops at state q_{Reject}, and outputs a 0 on the tape. When the input string is a palindrome, the machine eventually stops at state q_{Accept}, and outputs a 1 on the tape.

The input alphabet contains only two symbols: 0 and 1. The tape alphabet contains an additional symbol: the blank symbol B. The state transition table is shown in Table 3.12. Note that there are 9 states, but only seven states trigger state transitions. The machine stops when it enters any of the two final states. A final state does not trigger a state transition. In each of the seven states, the head may read one of three tape symbols 0, 1, or B. There are 3×7=21 transitions in Table 3.12.

Table 3.12 State transition table for a Turing machine to recognize any palindrome

Transition	Current State	Symbol Read	Symbol to Write	Head Move	Next State
1	q_0	0	B	$\rightarrow$	q_{Seen0}
2	q_0	1	B	$\rightarrow$	q_{Seen1}
3	q_0	B	1	$\leftarrow$	$\boldsymbol{q_{\text{Accept}}}$
4	q_{Seen0}	0	0	$\rightarrow$	q_{Seen0}
5	q_{Seen0}	1	1	$\rightarrow$	q_{Seen0}
6	q_{Seen0}	B	B	$\leftarrow$	q_{Want0}
7	q_{Seen1}	0	0	$\rightarrow$	q_{Seen1}
8	q_{Seen1}	1	1	$\rightarrow$	q_{Seen1}
9	q_{Seen1}	B	B	$\leftarrow$	q_{Want1}
10	q_{Want0}	0	B	$\leftarrow$	q_{Back}
11	q_{Want0}	1	B	$\leftarrow$	$q_{\text{BackErase}}$
12	q_{Want0}	B	1	$\leftarrow$	$\boldsymbol{q_{\text{Accept}}}$
13	q_{Want1}	0	B	$\leftarrow$	$q_{\text{BackErase}}$
14	q_{Want1}	1	B	$\leftarrow$	q_{Back}
15	q_{Want1}	B	1	$\leftarrow$	$\boldsymbol{q_{\text{Accept}}}$
16	q_{Back}	0	0	$\leftarrow$	q_{Back}
17	q_{Back}	1	1	$\leftarrow$	q_{Back}
18	q_{Back}	B	B	$\rightarrow$	q_0
19	$q_{\text{BackErase}}$	0	B	$\leftarrow$	$q_{\text{BackErase}}$
20	$q_{\text{BackErase}}$	1	B	$\leftarrow$	$q_{\text{BackErase}}$
21	$q_{\text{BackErase}}$	B	0	$\leftarrow$	$\boldsymbol{q_{\text{Reject}}}$

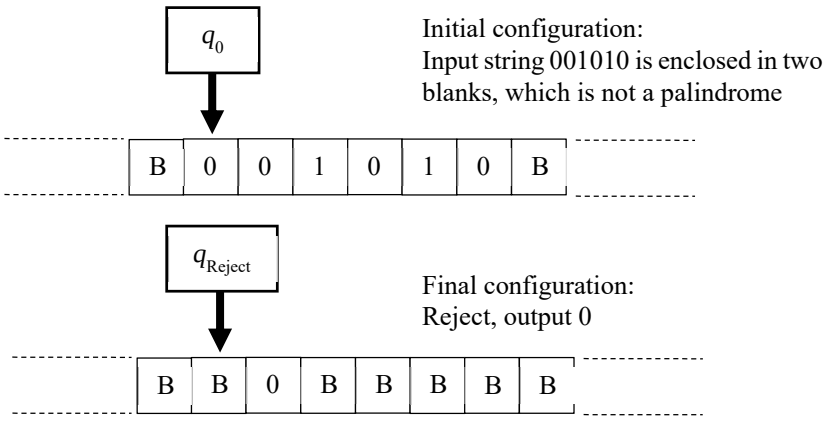
Figure 3.5 shows initial and final configurations of a Turing machine for recognizing palindromes, i.e., decides whether a string is a palindrome.

How does this Turing machine work? The basic idea is as follows.

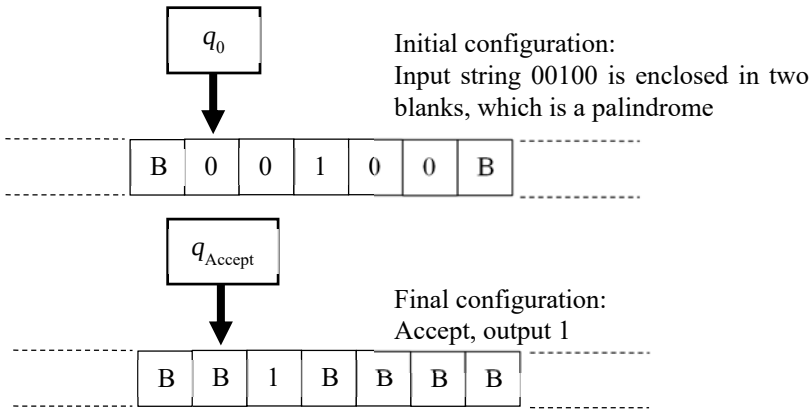
- Iterate over the given the input string.
  - When the first symbol and the last symbol match (they are both 0 or both 1), erase them and go to the next iteration.
  - When the first symbol and the last symbol do not match, erase the remaining string and enter q_{Reject}.
- Until all symbols of the input string are all erased, then enter q_{Accept}.

For instance, consider the input string 001010, which is not a palindrome. The tape configurations of the iterations are shown below:

- Iteration 1: …B**0**0101**0**B…; first and last symbols match, erase the two symbols and go to next iteration
- Iteration 2: …BB**0**10**1**BB…; first and last symbols do not match, reject.



(a) When the input string is 001010



(b) When the input string is 00100

Fig. 3.5 Initial and final configurations of a Turing machine for palindrome recognition

Now, consider the input string 00100, which is a palindrome. The tape configurations of the iterations are shown below:

- Iteration 1: …B**0**010**0**B…; first and last match, erase and go to next iteration
- Iteration 2: …BB**0**1**0**BB…; first and last match, erase and go to next iteration
- Iteration 3: …BBB**1**BBB…; one symbol matches itself, erase it and go to next iteration
- Iteration 4: …BBBBBBB…; all symbols erased, output 1 and enter q_{Accept}.
- Final result …BBB1BBBB…

Applying the Turing machine definition to the palindrome-recognition problem, we have the following rigorous and concrete definition: a Turing machine recognizing palindromes is a 7-tuple $M = \{Q, \Sigma, \Gamma, \delta, q_0, q_{\text{Accept}}, q_{\text{Reject}}\}$, where

- $Q = \{q_0, q_{\text{Accept}}, q_{\text{Reject}}; q_{\text{Seen0}}, q_{\text{Seen1}}, q_{\text{Want0}}, q_{\text{Want1}}, q_{\text{Back}}, q_{\text{BackErase}}\}$, the three states before the semicolon are special states common to many Turing machines: $q_0 \in Q$ is the initial state, $q_{\text{Accept}} \in Q$ is the accept state, and $q_{\text{Reject}} \in Q$ is the reject state.
- The input alphabet is $\Sigma = \{0,\ 1\}$.
- The tape alphabet is $\Gamma = \{0,\ 1,\ \text{B}\}$.
- The transition function $\delta: (Q - \{q_{\text{Accept}}, q_{\text{Reject}}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{\rightarrow, \leftarrow\}$ is defined by Table 3.12.

The machine starts at q_0, with the head points to the leftmost symbol of the input string. When a 0 or 1 is read, the head writes a blank to the pointed square and moves to the right, and the machine transition to state q_{Seen0} or q_{Seen1} which means the machine has seen a 0 or a 1. In such a state, the machine moves the head to the right, until it reads a B, indicating that the head has just passed the rightmost symbol (the end of the string). The machine then transitions to state q_{Want0} or q_{Want1}, indicating the machine is expecting a 0 or 1 from the end of the string, to match the 0 or 1 seen. If the head reads a matching 0 or 1 in q_{Want0} or q_{Want1}, the machine erases it by writing a B and enters state q_{Back}, to go back to the beginning of the string and start the next iteration. If the head reads a mismatching symbol, e.g., reading a 1 in state q_{Want0}, the machine enters $q_{\text{BackErase}}$ to erase all remaining input symbols, and then enters q_{Reject} and halts. If the head reads a B in q_{Want0} or q_{Want1}, the machine has erased all matching 0's and 1's, thus the machine enters state q_{Accept} and halts.

Note that in designing the state transition table, cares must be taken to ensure correct output, that is, a 1 is written on the tape when entering q_{Accept}, and a 0 is written on the tape when entering q_{Reject}.

Let us go through the step-by-step details of two small cases, for input strings 01 and 010, to verify the correctness of the Turing machine shown in Table 3.12.

For input string 01, which is not a palindrome, the sequence of transitions is shown in the following table, where each transition is a step in the computational process of deciding whether the input string is a palindrome. For each step, we list the tape contents before and after the transition, where the boldfaced symbol indicates the position of the read/write head. Before step 1, the tape contains B**0**1B and the head points to the square containing 0. The machine is at the initial state q_0, which triggers transition #1 in Table 3.12. After the transition, the symbol 0 is erased and the head moves right to point to the square containing 1.

Step	Before	Transition	After
1	B**0**1B	$< \#1, q_0, 0, B, \rightarrow, q_{\text{Seen0}} >$	BB**1**B
2	BB**1**B	$< \#5, q_{\text{Seen0}}, 1, 1, \rightarrow, q_{\text{Seen0}} >$	BB**1**B
3	BB**1**B	$< \#6, q_{\text{Seen0}}, B, B, \leftarrow, q_{\text{Want0}} >$	BB**1**B
4	BB**1**B	$< \#11, q_{\text{Want0}}, 1, B, \leftarrow, q_{\text{BackErase}} >$	B**B**BB
5	B**B**BB	$< \#21, q_{\text{BackErase}}, B, 0, \leftarrow, q_{\text{Reject}} >$	**B**0BB

For input string 101, which is a palindrome, the sequence of transitions is shown in the following table.

Step	Before	Transition	After
1	B**1**01B	$< \#2, q_0, 1, B, \rightarrow, q_{\text{Seen1}} >$	BB**0**1B
2	BB**0**1B	$< \#7, q_{\text{Seen1}}, 0, 0, \rightarrow, q_{\text{Seen1}} >$	BB0**1**B
3	BB0**1**B	$< \#8, q_{\text{Seen1}}, 1, 1, \rightarrow, q_{\text{Seen1}} >$	BB01**B**
4	BB01**B**	$< \#9, q_{\text{Seen1}}, B, B, \leftarrow, q_{\text{Want1}} >$	BB0**1**B
5	BB0**1**B	$< \#14, q_{\text{Want1}}, 1, B, \leftarrow, q_{\text{Back}} >$	BB**0**BB
6	BB**0**BB	$< \#16, q_{\text{Back}}, 0, 0, \leftarrow, q_{\text{Back}} >$	B**B**0BB
7	B**B**0BB	$< \#18, q_{\text{Back}}, B, B, \rightarrow, q_0 >$	BB**0**BB
8	BB**0**BB	$< \#1, q_0, 0, B, \rightarrow, q_{\text{Seen0}} >$	BBB**B**B
9	BBB**B**B	$< \#6, q_{\text{Seen0}}, B, B, \leftarrow, q_{\text{Want0}} >$	BB**B**BB
10	BB**B**BB	$< \#12, q_{\text{Want0}}, B, 1, \leftarrow, q_{\text{Accept}} >$	B**B**1BB

☲

21. Notable details of Turing machine

When learning Turing machines, students may experience several difficulties regarding details, which are summarized below.

Finite states. Any Turing machine has a finite number of states. Let us look at Table 3.12 again. The input string of palindrome can be of arbitrary length. However, the Turing machine has only 9 states. The same state transition table of 21 rows is used for input string of arbitrary length. It is a mistake to design a state transition table that depends on the length of the input string.

$\boldsymbol{B \notin \Sigma}$ **and** $\mathbf{B} \in \boldsymbol{\Gamma}$. The input blank symbol B belongs to the tape alphabet but does not belong to the input alphabet. It is a mistake to confuse the blank symbol B with the capital letter B (0x42), the ASCII Space symbol (0x20), or the ASCII Null symbol (0x00). When the input string needs to contain such symbols, we can change the blank symbol notation to a new symbol such as β. Also note that the read/write head points to a *tape* square, which contains a symbol in the tape alphabet, including all input symbols *and* the blank symbol.

No stop in the middle. The Turing machine stops (halts) only when it enters a final state, either q_{Accept} or q_{Reject}. If it is at a non-final state, a transition will always be triggered and the machine will enter the next state, which could be the same state as the current one. However, the machine will never stop at a non-final state. The reason is that by the Turing machine definition, the transition function δ is a mathematical *function*, which means that δ is defined for every element of $(Q - \{q_{\text{Accept}}, q_{\text{Reject}}\}) \times \Gamma$. That is, for every non-final state *s* and tape symbol *t*, $\delta(s,t)$ is always defined, and there is always a next state to transition to.

To design a Turing machine for some specific computing problem, it is usually more intuitive for the novice to draw state-transition diagram, e.g., Fig 3.4. However, one drawback of state-transition diagram is that it is easy to leave some $\delta(s,t)$ undefined. Though some transition seems impossible in the normal case, it is a good habit to write down the full transition function so as to handle the exceptional situations.

$(|\boldsymbol{Q}| - \mathbf{2}) \times |\boldsymbol{\Gamma}|$ **transitions**. It follows from the above discussion that the state transition table of a Turing machine will always have $(|Q| - 2) \times |\Gamma|$ rows of transitions, where $|Q|$ is number of elements of set *Q*. The value 2 is for the two final states q_{Accept} and q_{Reject}. For example, the Turing machine in Table 3.12 has 9 states and its tape alphabet Γ has 3 elements. Thus, its state transition table has (9-2)×3=21 rows. Note that, the calculation only works if both accept state and reject state exist in the Turing machine.

Explicit and implicit input/output. For any Turing machine, the input string must explicitly appear in the tape between two blanks, before any step of state transition happens. The output of the computation is often defined as the string between the head-pointed square and the first blank right of it. Sometimes, we more carefully and explicitly define the output. For instance, in Example 27, we define the output to be a single-symbol string 1 for q_{Accept}, and string 0 for q_{Reject}.

One may also simplify the situation by doing the computation without cleanup, but assuming implicit output instead. In such a case, the output string may be mixed with a subset of symbols of input strings and intermediate results.

3.3. Power and Limitation of Computing

Real world problems can be either abstract (e.g., mathematic problems) or concrete (e.g., the problem of searching the Web). These problems can be formulated as computational problems for Turing machines.

Most of the problems one can imagine can be solved by Turing machines. For example, adding two integers, deciding whether an integer is a prime number, finding the most economic routes for a traveling salesman, etc. However, there exist problems that cannot be computed by any Turing machine. Some problems cannot even be effectively expressed for a Turing machine to solve. The computer science field has encountered paradoxes, incomputable problems, and incompleteness results. **Computability** is the subfield of computer science that studies the power and limitation of computers.

The existence of incomputable problems seems to be a negative fact. However, people have found ways to exploit such negative results for positive benefits. The following are some examples of ideas:

- Incomputable problems provide opportunities for human intelligence.
- Computationally hard problems can be used to design computer and Internet games.
- If a privacy protection technique can be formulated as incomputable problems, one cannot use computers alone to break privacy protection.
- In his recent book *Life after Google: The Fall of Big Data and the Rise of the Blockchain Economy*, the futurist and industry analyst George Gilder suggests that incomputability results by Kurt Gödel and Alan Turing provide a foundational piece for future technology systems.

3.3.1. Church-Turing Hypothesis

We have an important positive result called Church-Turing Thesis, due to Alonzo Church (1903-1995) and Alan Turing (1912-1954). Because it is actually a hypothesis, not a fully proven statement, it is also called Church-Turing Hypothesis. The thesis says that no reasonable abstract computer is more powerful than Turing machines. More specifically, we have the following results.

A problem is **Turing computable**, if there is a Turing machine that correctly solves the problem. That is, for any given input string, the Turing machine starting at the initial state q_0 will correctly stop at q_{Accept} or q_{Reject}.

We say a problem is a **computable problem**, if it is Turing computable. In other words, Turing machines are a general-purpose model for computability. If a problem is Turing incomputable, no other reasonable abstract computer can solve the problem, either. The generality statement that

Computable = Turing computable

can be viewed as a definition supported by many proven results.

Church-Turing Thesis: Assume a reasonable abstract computer X is given. Any problem computable in X is also Turing computable.

We say X is **reducible** to Turing machines. Church, Turing and other scholars have proven that many powerful models of computation are reducible to Turing machines. Their main method is to treat a problem as a mathematical function and simulate a step of abstract computer X by Turing machine steps.

A more recent result is the so-called **Polynomial Church-Turing Thesis**: If a problem is computable in abstract computer X and costs T_x steps, it is computable in a Turing machine and costs T_t steps, such that $T_t = n^k \times T_x$. Here, *n* is the problem size and *k* is a constant. The Turing machine is at most polynomial times slower.

Consider the von Neumann model introduced in Chapter 2, which is a model of real computers such as a laptop computer. The von Neumann model can be augmented with infinite memory to obtain this result: **Turing machines are as powerful as a von Neumann computer** with infinite memory and arithmetic, logic, load, store, and conditional jump instructions. They can simulate each other with an overhead of no more than n^4.

3.3.2. (***) Incomputable Problems and Paradoxes

Computer science research also produced some seemingly negative results of computability. We discuss two such incomputable problems.

The halting problem. Given the description of an arbitrary Turing machine *M* and an input string *x*, decide whether *M* will terminate or run forever. "Turing machine terminates" means it eventually enters the accept state or the reject state.

The Entscheidungsproblem (the decision problem). Given a real number, decide if it is Turing computable. That is, if there is a Turing machine which can write down arbitrarily long decimal digits of the real number. If one wants *n* digits, for any *n*, the Turing machine will output the correct *n* digits and stop.

In this section, we show why the halting problem is not computable in Turing machine, and leave the Entscheidungsproblem as a thinking problem.

Example 28. The halting problem is not Turing computable

We firstly give an example to show the case where a Turing machine may not terminate for some input string. Let us modify the machine in Fig. 3.4 and Table 3.11 a little bit. See Table 3.13 for the modified description of the state transition.

Table 3.13 State transition table of a Turing machine

Current State	Symbol Read	Symbol to Write	Head Move	Next State
q_0	0	B	→	q_0
q_0	1	B	→	q_0
q_0	@	B	→	q_1 (Halt)
q_0	B	B	→	q_0

In this Turing machine, if the input string is 0000101@ as illustrated in Fig 3.4, the machine will terminate in the halt state after reading the symbol @. However, if the input string is 0000101, the machine will never terminate and be stuck in state q_0. Thus, for the halting problem, if the input is this Turing machine and input string 0000101@, the answer should be "YES" or 1; while if the input string is 0000101, the answer should be "NO" or 0.

Before discussing the halting problem, let us first look at the representation of Turing machine more carefully. Any Turing machine can be represented by a 7-tuple $M = \{Q, \Sigma, \Gamma, \delta, q_0, q_{\text{Accept}}, q_{\text{Reject}}\}$, so it can be represented by a finite binary string. For example, we can write down the Turing machine in the normal way, like Q={q0,q1,q2}, and then translate it to ASCII code which is a finite binary string. Note, in such representation, not every finite binary string corresponds to a Turing machine, but it is easy to design a Turing machine which can decide whether a finite binary string corresponds to a Turing machine or not.

Thus, the set of all Turing machines is a subset of the set of all finite binary strings. This means the set of all Turing machines is countable. So is the set of all possible input strings.

We will prove the halting problem is not Turing computable by contradiction. Suppose there exists some Turing machine H which can compute halting problem. That is, for any Turing machine M, and for any input string x, $H(M, x) = 1$ or stops in the accept state if Turing machine M will terminate with the input string x; and $H(M, x) = 0$ or stops in the reject state if Turing machine M will never terminate (run forever) with the input string x. Thus H actually computes the following matrix in Fig 3.6.

	1	2	3	4	5	6	7	8	9	...
1	0	0	0	0	0	0	0	0	0	...
2	1	1	1	1	1	1	1	1	1	...
3	0	1	0	0	0	0	0	0	0	...
4	0	0	1	1	1	1	1	1	0	...
5	1	0	0	0	0	0	0	0	0	...
6	1	1	1	1	1	1	1	1	1	...
7	1	1	0	0	1	1	1	1	1	...
...	.	.	.	.	.	.	.	.	.	.

Fig. 3.6 Illustration of the halting problem

In this matrix, the i-th row represents the Turing machine whose binary representation is i, and the j-th column represents the input string j. If the Turing machine i can terminate with the input string j, the element of i-th row and j-th column in the matrix is 1; and if the Turing machine i cannot terminate with the input string j, the element of i-th row and j-th column in the matrix is 0. For some binary representation i, if there is no Turing machine corresponding to i, we set the i-th row to be all 1. Note that, in this matrix, we can list all possible Turing machines and all possible input strings. Of course, the matrix is infinitely large, with infinite number of rows and columns. But the number of rows and columns are both countable since the number of Turing machines and input strings are countable. Thus, the function of Turing machine H is actually to compute such a matrix.

Now, let us define a new Turing machine G based on H as follows.

```
Turing machine G
Input string: binary string i
        Run H(i, i);
        If H(i, i) = 0,
                Then, go to the accept state and halt
                Otherwise, run forever
```

Fig. 3.7 The definition of Turing machine G

Since H exists, we can construct such Turing machine G. Now let us consider the case when the input string is G, the binary representation of the Turing machine G. For simplicity, we will use G to indicate both the Turing machine and its binary representation.

Consider the element in the G-th row and G-th column in the matrix in Fig 3.6. If the element is 1, it means two things: (1) when the Turing machine G takes the input string G, it should halt after finite steps. This is due to the definition of the matrix. (2) $H(G, G) = 1$ due to the definition of H. However, examine the definition of G, we know when it takes input string G, it will run $H(G, G)$. Since $H(G, G) = 1$, the Turing machine G will run forever and never halt. Contradiction.

The other case is similar. If the element in the G-th row and G-th column in the matrix is 0, it also means two things: 1) when the Turing machine G takes the input string G, it should never halt. This is due to the definition of such matrix. 2) $H(G, G) = 0$ due to the definition of H. However, examine the definition of G, we know when it takes input string G, it will run $H(G, G)$. Since $H(G, G) = 0$, the Turing machine G will go to the accept state and halt. Contradiction again.

The element must take 0 or 1, and both cases will lead to contradiction. Thus, the assumption we made is not true. That is, there does not exist some Turing machine H which can compute the halting problem. Equivalently, the halting problem is not Turing computable.

☶

One might think that the reason incomputable problems exist is because Turing machines are not powerful enough. There may exist other more powerful computational model which might be able to solve the halting problem or the decision problem. However, Church-Turing Hypothesis tells us this is not the case. From this viewpoint, Church-Turing Hypothesis is a negative result.

Some problems cannot even be effectively expressed as computational problems for Boolean logic or Turing machines to solve. A class of such problems are called paradoxes. We discuss two paradoxes below. The two cases involve self-reference and self-contradiction.

When encoding a real-world problem into the cyberspace, we need to be aware of paradoxes. We often need to change the problem specification to avoid any paradox. Sometimes, we can utilize paradox to create new functionality. An example of utilizing self-reference will be discussed in Section 5.3.2.

The liar paradox. When a liar says "This sentence is false", is he telling the truth? Remember the impatient guide problem in Examples 13 and 18. Suppose the guide comes from the Lying Village and makes the statement "What I am saying is false." Is the guide lying or not?

Such paradoxes contain a strange expression when expressed in Boolean logic. Let X stands for X=false. Sometimes we denote this naming as X: X=false. Now, is X true or false?

More generally, we may define Boolean variables using self-referencing expressions. An example is a pair of self-referencing expressions $\mathrm{Q} = \mathrm{Q} + \mathrm{S}$ and $\overline{Q} = \overline{Q} + \mathrm{R}$. In Section 5.3.2, we will see that these expressions with feedbacks are partial specification of the S-R latch, a new functionality to represent states. The contradiction is reconciled, when Q before and Q after the equality sign represent the next state and the current state, respectively.

The barber paradox. There is only one barber in a village who shaves all in the village who do not shave themselves. Does the barber shave himself?

A mathematic version of the barber paradox is **Russell's paradox** discovered by British scholar Bertrand Russell. In set theory, Russell's paradox considers the set R of all sets that are not members of themselves. Is R a member of itself? If we answer Yes (R is a member of itself), by the definition of R, R is NOT a member of itself. If we answer No (R is not a member of itself), by the definition of R, R must be a member of itself, since R is the set of **all** sets that are not members of themselves. Thus, whether we answer Yes or No, a contradiction will be the result.

Russell's paradox has contributed to the foundation of mathematics. In particular, it stimulated the creation of modern set theory (Zermelo–Fraenkel set theory), which is also a foundation of modern computer science.

3.3.3. (***) Gödel's Incompleteness Theorems

In 1928, David Hilbert proposed a suggestion for the solution to the foundational crisis of mathematics: to establish a set of axiom systems so that all mathematical propositions can be shown to be true or false in this system within a limited number of steps. The system Hilbert envisioned needs to answer the following questions:

- **Completeness:** for each of the true mathematical statements, we should be able to give a proof in this system.
- **Consistency:** there is no contradiction in the system, that is, there will be no statement that we can prove to be true and to be false in the same system.
- **Decidability:** we can find a way to determine whether a mathematical statement is true or false through only "mechanized" deduction.

For the general decidability question, i.e., the Entscheidungsproblem (the decision problem), Alan Turing's 1936 paper gave a negative answer.

How about the completeness and the consistency questions? In 1931, only three years after Hilbert's suggestion, Kurt Gödel (1906-1978) gave negative answers to these questions. The negative answers are called Gödel's incompleteness theorems.

Gödel's first incompleteness theorem: Any mathematical system that includes elementary number theory (natural numbers, addition, and multiplication) cannot have completeness and consistency at the same time.

Usually, researchers choose to sacrifice completeness in this dilemma. That is to say, for any reasonably sophisticated mathematical system, there exists some statement that is true, but we cannot prove it in this system. Some researchers have suggested that the "Goldbach's conjecture" may be the case. Here "reasonably sophisticated" means including the elementary number theory known as Peano Arithmetic shown in Box 9.

Box 9. Peano's Axioms of Arithmetic

In 1889, Giuseppe Peano (1858–1932) proposed Peano's axioms of arithmetic, later also called **Peano Arithmetic** for simplicity. This result has been widely used since then for mathematic logic in general and elementary number theory in particular. At a minimum, Peano Arithmetic consists of the following five axioms, and addition and multiplication operators can be defined based on these axioms.

1. Zero is a natural number; $0 \in \mathbb{N}$.
2. Every natural number has a successor in the set of natural numbers; $\forall n \in \mathbb{N}\ [\mathrm{S}(n) \in \mathbb{N}]$.
3. Zero is not the successor of any natural number; $\forall n \in \mathbb{N}\ [\mathrm{S}(n) \neq 0]$.
4. If the successors of two natural numbers are the same, then the two original numbers are the same; $\forall m, n \in \mathbb{N}\ [\mathrm{S}(m) = \mathrm{S}(n) \rightarrow m = n]$.
5. If a set contains zero and the successor of every natural number, then the set contains the set of natural numbers. This is called the induction axiom.

Gödel's second incompleteness theorem: For any mathematical system that includes elementary number theory, if it is consistent, then its consistency cannot be proved within itself.

In other words, whether there is a paradox in the system cannot be solved by relying on this system alone.

Gödel's incompleteness theorems deny Hilbert's proposal. Gödel's first incompleteness theorem tells us that truth and provability are two different things. A provable statement must be true if we stick to consistency, but a true statement is not necessarily provable since we sacrifice completeness. Gödel's second incompleteness theorem tells us that consistency cannot be proved within a mathematical system itself.

Example 29. A case of Gödel's first incompleteness theorem

We discuss a specific statement to make our understanding of Gödel's first incompleteness theorem more concrete. The statement is Goodstein theorem, which is true, but cannot be proven in any mathematical system that includes elementary number theory.

We first need the concept of a Goodstein sequence.

Given any natural number *n*, its *hereditary base-b representation* is obtained as follows. First, write the sum of power base-*b* representation of *n*. If some exponent m is greater than b, replace *m* by *m*'s own sum of power base-*b* representation. Repeat this process until all numbers are less than or equal to b.

For instance, given *n*=266, its base-2 representation is $2^8 + 2^3 + 2$. There are two numbers >2, i.e., 8 and 3. Replacing them by their base-2 representations, we have $2^{2^3} + 2^{2+1} + 2$. Now we have only one number >2, which is 3. Replacing 3 with its base-2 representation, we have $2^{2^{2+1}} + 2^{2+1} + 2$. This is the hereditary base-2 representation of 266.

The change-of-base function $R_b(n)$ changes b to b+1 in *n*. That is, $R_b(n)$ replaces every b with b+1 in the hereditary base-b representation of *n*. For instance, for *n*=266, the hereditary base-2 representation is $2^{2^{2+1}} + 2^{2+1} + 2$. Changing every occurrence of 2 to 3, we have $R_2(266) = 3^{3^{3+1}} + 3^{3+1} + 3$, which is in the form of a hereditary base-3 representation of a much larger number:

$$R_2(266) = 443426488243037769948249630619149892887.$$

The *Goodstein sequence* for a given natural number *n* is denoted as $(n)_k$, where k ranges over the set of natural numbers, and the value of each $(n)_k$ is written as follows:

$$(n)_0 = n$$
$$(n)_1 = R_2(n) - 1$$
$$(n)_2 = R_3((n)_1) - 1$$
$$\dots$$
$$(n)_{k+1} = \begin{cases} R_{k+2}((n)_k) - 1 & \text{if } (n)_k > 0 \\ 0 & \text{if } (n)_k = 0 \end{cases}$$

For instance, for *n*=266, the Goodstein sequence is

$$(266)_0 = 2^{2^{2+1}} + 2^{2+1} + 2 \qquad = 266$$
$$(266)_1 = 3^{3^{3+1}} + 3^{3+1} + 2 \qquad \approx 4.4 \times 10^{38}$$
$$(266)_2 = 4^{4^{4+1}} + 4^{4+1} + 1 \qquad \approx 3.2 \times 10^{616}$$
$$\dots$$

A Goodstein sequence has three noteworthy properties:

- To go from one number to the next, two operations are performed. First, apply the change-of-base function, then subtract 1 from the result. The change-of-base function seems to significantly increase the number.
- The sequence grows tremendously fast. For instance, going from $(266)_1$ to $(266)_2$, the number grows to 3.2×10^{616}. Compare this to the fact that there are only about 10^{90} basic particles in the observable universe.
- The striking property is that this quickly growing sequence approaches 0! This is Goodstein theorem.

Goodstein theorem: Every Goodstein sequence always approaches 0.

That is, for any natural number *n*, there exists another natural number *m*, such that $(n)_m = 0$.

In 1944, Reuben Goodstein proved the Goodstein theorem. In 1982, Laurie Kirby and Jeff Paris showed that Goodstein theorem cannot be proven in any mathematical system that includes elementary number theory (i.e., Peano's Arithmetic).

Let us consider $(4)_k$ in more detail, i.e., the Goodstein sequence for number 4. Goodstein's function $G(n) = m$ is a function that maps n to m, where m is the smallest natural number such that $(n)_m = 0$. It is known that

$$G(4) = 3 \times 2^{402653211} - 3 \approx 6.895 \times 10^{121210694}.$$

It is impractical to compute all the non-zero items of the sequence. Our TA, Hongrui Guo, computed the first five, the largest five, and the last five items of the sequence before it reaches 0. These 15 values are as the following.

$$(4)_0 = 2^2 = 4$$
$$(4)_1 = 3^3 - 1 = 2 \times 3^2 + 2 \times 3^1 + 2 = 26$$
$$(4)_2 = 2 \times 4^2 + 2 \times 4^1 + 2 - 1 = 41$$
$$(4)_3 = 2 \times 5^2 + 2 \times 5^1 + 1 - 1 = 60$$
$$(4)_4 = 2 \times 6^2 + 2 \times 6^1 - 1 = 2 \times 6^2 + 6 + 5 = 83$$
$$\dots$$
$$(4)_{max-2} \approx \text{1st1000DigitsOfMax}$$
$$(4)_{max-1} \approx \text{1st1000DigitsOfMax}$$
$$(4)_{max} \approx \text{1st1000DigitsOfMax}$$
$$(4)_{max+1} \approx \text{1st1000DigitsOfMax}$$
$$(4)_{max+2} \approx \text{1st1000DigitsOfMax}$$
$$\dots$$
$$(4)_{G(4)-4} = 4$$
$$(4)_{G(4)-3} = 3$$

$(4)_{G(4)-2} = 2$
$(4)_{G(4)-1} = 1$
$(4)_{G(4)} = 0$

The sequence $(4)_k$ reaches the maximal value at index *max*. The maximal value is $(4)_{max} = 3.44754040154631\ldots. \times 10^{121210695}$. It is a natural number with 121210695 decimal digits. The first 1000 decimal digits of the largest numbers in the sequence are:

344754040154631008286819497980575497847887493790148682948258247118
137174798986243594412653772313883635770634387067098147137377012311
972582711710423708488689955731916776345646600366175225653655667076
607366382215502787249662525750333088534866784863341149319031461526
965596973686649216094822529043684736588670987614756209204200586866
497331182917585633213812021951719841820181233533930105627134871122
877429506752901948699802511108360780114527916992721682291424810789
456193340854410358943085149505243047152149159691566503117689965161
095721221736078065615470715884693378579337518896782222962282279777
804376115277338671809923516166212703892541980539479298098193948648
552290922287885788388756034836731638127067328067534738276921901537
543261656510810808181431092371120331330596839997167695778121779569
475400362539158893903885373347987634477272363235750176209299371955
055294414557410495717377709254930927728668041323883134245214544951
623092744525525577131065227475935299300530660600856297317088013008
9684337752.

☷

3.4. Exercises

1. What is NOT a possible truth value of proposition formula $P \vee Q$?

   (a) 0
   (b) 1
   (c) Either 0 or 1
   (d) Both 0 and 1

2. What is NOT a possible truth value of proposition formula $(P \vee \neg Q) \rightarrow P$?

   (a) 0
   (b) 1
   (c) Either 0 or 1
   (d) Both 0 and 1

3. Let the proposition formula G be $P \rightarrow Q$. How many assignments of the truth value to P, Q are there to make G **false**?

   (a) 1
   (b) 2
   (c) 3
   (d) 4

4. Let the proposition formula G be $(\neg Q \vee R) \leftrightarrow (\neg P \wedge R)$. How many assignments of the truth value to P, Q, R are there to make G **false**? Here, $A \leftrightarrow B$ is defined as $(A \rightarrow B) \wedge (B \rightarrow A)$

   (a) 2
   (b) 3
   (c) 4
   (d) 5

5. Write down the truth table of two proposition formulae $P \rightarrow \mathrm{Q}$ and $\neg P \vee Q$ and show that they are equivalent formulae.
6. Write down the disjunctive normal form of the two proposition formulae $P \vee Q$ and $P \wedge Q$.
7. Write down the disjunctive normal form of the proposition formula $P \vee \neg P$.
8. The conjunctive normal form of proposition formula $\neg(P \rightarrow Q)$ is

   (a) $(P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee \neg Q)$
   (b) $P \vee \neg Q$
   (c) $(P \vee Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q)$
   (d) $P \wedge \neg Q$

9. In the theorem of disjunctive normal form, why do we need the assumption $F(x_1, x_2, \ldots, x_n) \not\equiv 0$?
10. Which of the following formula is not a tautology? Tautology refers to the proposition formula that is **true** in every possible assignment.

(a) $(P \oplus P) \leftrightarrow (Q \wedge \neg Q)$
(b) $(P \oplus \neg P) \leftrightarrow (Q \vee \neg Q)$
(c) $((P \vee Q) \rightarrow P) \leftrightarrow (R \rightarrow R)$
(d) $((P \wedge Q) \rightarrow P) \leftrightarrow (R \rightarrow R)$

11. Which of the following formula is a tautology?

(a) $(P \rightarrow Q) \leftrightarrow (Q \rightarrow P)$
(b) $(P \rightarrow Q) \leftrightarrow (\neg Q \rightarrow \neg P)$
(c) $(P \rightarrow Q) \leftrightarrow (\neg Q \rightarrow P)$
(d) $(P \rightarrow Q) \leftrightarrow (Q \rightarrow \neg P)$

12. Which of the following equation about "exclusive or" is correct?

(a) $(x \oplus y) \wedge z = (x \wedge z) \oplus (y \wedge z)$
(b) $(x \oplus y) \vee z = (x \vee z) \oplus (y \vee z)$
(c) $\neg(x \oplus y) = (\neg x) \oplus (\neg y)$
(d) $(x \vee y) \oplus z = (x \oplus z) \vee (y \oplus z)$

13. How many different Boolean functions of 4 variables are there?

(a) 16
(b) 32
(c) 65,536
(d) 4,294,967,296

14. Every playing card has two sides. One side is a number and the other side is a letter. Now there are four cards on the table, with A, 3, S, 8 facing up. **In the worst case**, how many cards do you need to turn over to confirm whether the following proposition is true for these four cards: the number on the vowel card (cards with letters AEIOU) must be even.

(a) 3
(b) 2
(c) 1
(d) 4

15. Three people, Alice, Bob and Charlie, said the following three sentences.

- Alice: Both Bob and Charlie lie.
- Bob: I tell the truth.
- Charlie: Bob lies.

Which of the following choices must be true?

(a) Charlie lied.
(b) Alice lied.
(c) Bob lied.
(d) All the previous 3 choices may be false.

16. Is the following logic correct? That is, assuming that the premise is true, is the conclusion also true? Please explain your answer.

- Premise (1): students who take the course of Introduction to Computer Science can master Golang.
- Premise (2): Some students who master Golang can serve as the teaching assistants in the course of Introduction to Computer Science next year.
- Conclusion: some students who take the course of Introduction to Computer Science can serve as teaching assistants next year.

17. Denote by P the statement "I will travel around the world" and Q the statement "I have enough money". Let f be the statement "I will travel around the world, only if I have enough money". Which is the correct symbolization of f?

(a) $Q \rightarrow P$
(b) $P \rightarrow Q$
(c) $P \leftrightarrow Q$
(d) $\neg$P $\vee$ $\neg$Q

18. Let $P(x)$ denote the statement "x masters Golang", $Q(x)$ denote the statement "x take the course of Introduction to Computer Science", and $R(x)$ denote the statement "x can serve as teaching assistants in the course of Introduction to Computer Science next year". Let f be the statement "students who take the course of Introduction to Computer Science can master Golang" and g be the statement "some people who master Golang can serve as teaching assistants in the course of Introduction to Computer Science next year". Which is the correct symbolization of f and g?

(a) $f: \forall x\big(Q(x) \wedge P(x)\big); g: \exists x(P(x) \wedge R(x))$
(b) $f: \forall x\big(Q(x) \rightarrow P(x)\big); g: \exists x(P(x) \rightarrow R(x))$
(c) $f: \forall x\big(Q(x) \rightarrow P(x)\big); g: \exists x(P(x) \wedge R(x))$
(d) $f: \forall x\big(Q(x) \wedge P(x)\big); g: \exists x(P(x) \rightarrow R(x))$

19. Can a Turing machine stops in the middle? Select the correct answer.

(a) No. A Turing machine stops when it enters an accept state or a reject state.
(b) Yes. A Turing machine can stop before it enters an accept state or a reject state, because the head sees a symbol not recognizable.
(c) Yes. A Turing machine can stop before it enters an accept state or a reject state, because the machine enters a state with no next state.
(d) Yes. A Turing machine can stop before it enters an accept state or a reject state, because the state transition table has only a finite number of entries, which cannot model all possible state transitions.

20. Design a Turing machine to accept the language $L = \{0^n | n \geq 1\}$ where the input alphabet is $\Sigma = \{0, 1\}$ and B represents the blank symbol. That is, the Turing machine should accept 0 or 000, but reject 010 or 100.

21. Design a Turing machine to accept the language $L = \{0^a 1^b 2^c | a, b, c \geq 0, a + b = c\}$ where the input alphabet is $\Sigma = \{0, 1, 2\}$. That is, the Turing machine should accept 0122 or 02, but reject 012 or 1002.
22. (***) In the definition of Turing machine, if the transition function is specified as $Q \times \Gamma \to Q \times \Gamma \times \{\to\}$, which means that the Turing machine can only move its head to the right and cannot move its head to the left in each state, we call it a **right-moving Turing machine**. Which of the following propositions about right-moving Turing machine and Turing machine is correct?

    (a) There is a computing task which can be decided by Turing machine, but not by right moving Turing machine.
    (b) There is a computing task which can be decided by the right moving Turing machine, but not by Turing machine.
    (c) Right moving Turing machine and Turing machine have the equivalent computing power.
    (d) None of the above three propositions has been proved at present.

23. (***) Use the Pumping Lemma to prove that palindromes cannot be recognized by finite automata.

We say that an automaton recognizes the language of all palindromes, if when a character string of finite length is fed to the automaton as input, the automaton will finish in finite number of steps and output 1 if the string is a palindrome, and 0 if the string is not a palindrome.

Pumping Lemma. Let L be a language recognized by finite automata. Then there exists an integer n depending only on L such that every string $w \in L$ of length at least p (called the "pumping length") can be written as $w = xyz$ (w can be divided into three substrings), satisfying the following conditions:

a) $|y| \geq 1$
b) $|xy| \leq p$
c) $\forall k \geq 0, xy^k z \in L$

24. Regarding the Church-Turing Hypothesis, which of the following is correct?

    (a) The Hypothesis shows the generality feature of logic thinking. It says that Turing machine is a general-purpose model of computation.
    (b) The Hypothesis says that Turing machine is not as general purpose as my laptop computer, because a Turing machine cannot create a PowerPoint presentation file.
    (c) The Hypothesis says that Turing machine is general-purpose. Thus, one can use Turing machine to automatically prove the Goodstein theorem.
    (d) The Hypothesis says that Turing machine and my laptop computer have equal power, in terms of computability.

3.5. Bibliographic Notes

The chapter quotation is from Professor Georg Gottlob of Oxford University, in a keynote speech addressing the 2009 European Computer Science Summit [1]. Kleene logic and the number of Kleene expressions are discussed in [2]. Kirby and Paris showed that Goodstein's theorem [3] cannot be proven in a mathematical system containing Peano Arithmetic [4].

[1] Gottlob, G. (2009). Computer science as the continuation of logic by other means. Keynote Address, European Computer Science Summit.

[2] Mukaidono, M. (1982). New canonical forms and their application to enumerating fuzzy switching functions. In Proc. 12th Int. Symp. on Multiple-Valued Logic (pp. 275-279).

[3] Goodstein, R. L. (1944). On the restricted ordinal theorem. The Journal of Symbolic Logic, 9(2), 33-41.

[4] Kirby, L., & Paris, J. (1982). Accessible independence results for Peano arithmetic. In Bulletin of the London Mathematical Society, 14, 285-293.