# Contents

1 Why study discrete structure?  

2 Propositional logic  
   2.1 Statement v.s. proposition  
   2.2 Logical operators  
      2.2.1 Negation, And, Inclusive Or  
      2.2.2 Exclusive Or  
      2.2.3 Imply  
      2.2.4 If and only if  
   2.3 Converse, contrapositive, and inverse  
   2.4 Priorities of logical operators  
   2.5 De Morgan’s Laws and other logical equivalences  
   2.6 Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF)  

3 Predicate logic  
   3.1 Quantifiers  
      3.1.1 Universal quantifier $\forall$  
      3.1.2 Existential quantifier $\exists$  
      3.1.3 Priorities of quantifiers  
      3.1.4 De Morgan’s laws in predicates  
   3.2 Translation from English to predicates  
   3.3 Nested quantifiers  

4 Proof and disproof  
   4.1 Basic principal of proof  
   4.2 Proof by contradiction  
      4.2.1 A sample proof by contradiction  
   4.3 Proof by induction  
      4.3.1 A sample proof by induction  
   4.4 Disproof
Acknowledgement

All of my lecture notes are made with the lecture notes given by Professor Edward M. Reingold and Professor Xiang-Yang Li, and many of their examples are used in making my lecture notes. I want to express my sincere gratefulness to them for sharing the teaching materials they have created.

1 Why study discrete structure?

I’d like to start my answer with an answer to a more fundamental question: ‘what is Computer Science’? I always thought ‘Computer Science’ is a bad name to describe what we learn and research in this major because the ‘Computer’ in C.S. is not a laptop nor a desktop. What is the ‘Computer’ in ‘Computer Science’ then?

In fact, the ‘computer’ in computer science (in most cases) refers to this mathematical model of the device who has an ability to ‘compute’:

\[ \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle \]

which is called Turing Machine. We will learn a simple version of it at the end of the semester (Finite State Machine), and let’s first ignore those fancy notations at the moment. The point is, what we computer scientists (Yes! You are a scientist as well!) study is the science of solving theoretic problems by using the device that can compute – computer.

By coincidence, there ‘happens to be’ devices that are the realization of the above computing model (i.e., laptop/desktop/smartphones), and those devices ‘happen to’ became proliferation in our life now. That is why people started to do cool things and solve real life problems using laptops/desktops/smartphones, and that is why most outsiders think Computer Science is a study of solving problems using laptops.

Instead, what we mostly do is the abstraction of theoretic problems from the real-life problems, and after we have come up with theoretic problems, we study how we can solve the theoretic problems in a better way.

Now I’m ready to answer our initial question: why do we need to study discrete structure? We need to solve the mathematical problems using the turing machine, and that machine we are going to use to implement our solutions and ideas is built almost entirely on discrete mathematics. Therefore, in order to fundamentally study the science about ‘computer’ (i.e., Computer Science), the discrete structure is one of the ‘must-do’s before you step deeper into the area.

2 Propositional logic

Logic is a formal form of arguments, which is often used to formally describe certain statements, and it is used to describe the protocols as well in computer science. Its applications in computer science include, but not limited to

- Software engineering
  - Validate the correctness of the software specifications.
  - Transform the specifications into efficient code on diverse platforms and prove the equivalence between the implementation and the specifications.
  - Prove the correctness of procedures and estimate the number of steps required to execute a specified program.
• Automatic security proof
  – Formally define the protocol
  – Automatically prove that the behavior of the protocol is within the safe states.

Logic is a huge family of different types of logic languages, and let us start with the simplest one – propositional logic.

2.1 Statement v.s. proposition

A statement is a declarative sentence, and a proposition is a declarative sentence that is either true or false. Only objective statements are propositions because one cannot answer yes or no deterministically without ambiguity to subjective statements, and this is what makes a proposition different from a statement. For example,

‘Are you a student?’, ‘Say yes!’

are not statements because they are not declaring anything.

‘I am tall’, ‘Chicago is far from San Francisco’, ‘You cannot do that’

are statements because they are declaring something, but they are not propositions because ‘being tall’, ‘being far’, and ‘cannot do that’ are subjective, and one cannot determine the truth non-ambiguously. However,

‘I am taller than Chris who is 175cm’, ‘Chicago is further from San Francisco than from Hawaii’

are propositions because one can determine whether the statements are true without ambiguity.

2.2 Logical operators

Logical operators are used to describe the relationship of propositions in compound propositions. Commonly used ones include

<table>
<thead>
<tr>
<th>Table 1: Operators in the order of priority (high to low)</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬</td>
</tr>
<tr>
<td>∧</td>
</tr>
<tr>
<td>∨</td>
</tr>
<tr>
<td>⊕</td>
</tr>
<tr>
<td>→</td>
</tr>
<tr>
<td>↔</td>
</tr>
</tbody>
</table>

Note that ⇒ is different from →, and ⇔ is different from ↔ as well.

2.2.1 Negation, And, Inclusive Or

The first three are easy to understand. However, there is a common mistake in the negation.

• ¬(All of the students are female) is not (None of the students are female).
  – It should be (Not all of the students are female) = (At least one of the student is not female).
2.2.2 Exclusive Or

\( \lor \) corresponds to the commonly used English sentence ‘either ... or ...’, and it is commonly observable in a programming language as well (e.g., C, Java). The English version of \( \oplus \) is ‘either ... or ... but not both’. That is why it is called as exclusive or. \( X \oplus Y \) is true when only one of \( X, Y \) is true, and it is false when both are true or both are false.

<table>
<thead>
<tr>
<th>( X )</th>
<th>( Y )</th>
<th>( X \oplus Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

The above table is called truth table of \( X \oplus Y \). A truth table describes what is the truth values of all combinations of T & F of the propositions.

2.2.3 Imply

\( \rightarrow \) corresponds to ‘if ... else ...’ in English, however the truth table of \( X \rightarrow Y \) may be counter-intuitive.

<table>
<thead>
<tr>
<th>( X )</th>
<th>( Y )</th>
<th>( X \rightarrow Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

It is easy to understand the first two rows, but the last two rows are quite counter-intuitive. One of the ways to interpret this logical operator is to consider the ‘if... then ...’ statements in the contract or the law. For example

- ‘If the tenant smokes, he/she will pay the cleaning fee $50’
  - If someone smokes but does not pay the fee, then the above statement is wrong. However, if someone did not smoke and paid the fee, or if someone did not smoke and did not pay the fee, it belongs to a different case that is not specified by the above statement, and it does not hurt the statement to say so. Therefore, the statement is correct.

- ‘If a person gets a speed ticket, he/she will pay the fine $150’
  - Similarly, the statement is correct (assuming the person will not choose to go to the court).

Our common sense also tells the above statements are correct, and this is also applicable in logical deduction as well, therefore we have \( F \rightarrow T = F \rightarrow F = T \).

Based on this observation, we can conclude the following propositions are weird in English or common sense but they are logically true.
• If the Moon is made of flowers, everyone in CS330 will get an A.
• If the instructor is the president of United States, the instructor will buy every student a doughnut.
• If our class is at LS 121, there will be no exam in CS330 this semester.

Specifically, when $X \rightarrow Y$ is always true, we also write it as $X \Rightarrow Y$.

### 2.2.4 If and only if

$X \leftrightarrow Y$ means $X \rightarrow Y$ and $Y \rightarrow X$, and in English it means ‘$X$ and $Y$ are logically equivalent’. Logical equivalence exists when two propositions always have the same truth value. For example, it is known that $(X \rightarrow Y) \leftrightarrow (\neg X \lor Y)$ because we have the following truth table. We can observe that the propositions

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$X \rightarrow Y$</th>
<th>$\neg X \lor Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

have the same truth values in all cases.

Specifically, when $X \leftrightarrow Y$ is always true, we also write it as $X \equiv Y$.

### 2.3 Converse, contrapositive, and inverse

From $X \rightarrow Y$, we can derive following variants.

• $Y \rightarrow X$: converse of $X \rightarrow Y$
• $\neg Y \rightarrow \neg X$: contrapositive of $X \rightarrow Y$
• $\neg X \rightarrow \neg Y$: inverse of $X \rightarrow Y$

One important fact in the logical reasoning is: $X \rightarrow Y$ and $\neg Y \rightarrow \neg X$ are logically equivalent, and the other two are not equivalent to $X \rightarrow Y$. Let’s see one example.

$X$: ‘Monday is holiday’, $Y$: ‘There is no class on Monday’

Obviously, $X \rightarrow Y$ is true in our course. Let’s see the other variants. The converse of $X \rightarrow Y$ is ‘If there is no class on Monday, Monday is holiday’, which is false because there is no class on July 11th but it is not a holiday (corresponding to $T \rightarrow F$). The inverse of $X \rightarrow Y$ is ‘If Monday is not holiday, there is a class on Monday’, which is false again because July 11th is not a holiday but there is no class on that day (corresponding to $T \rightarrow F$). However, the contrapositive of it is ‘If there is class on Monday, Monday is not holiday’, and it is true because it is not possible to find a case where the contrapositive translates to $T \rightarrow F$. 

2.4 Priorities of logical operators

Similar to arithmetic operators (+, −, ×, ÷, etc.), logical operators have different priorities as well. Part of them is listed in Table 1. It tells us that:

\[ X \lor Y \to Z \equiv (X \lor Y) \to Z \neq X \lor (Y \to Z) \]

Although standard priorities exist and there should be no ambiguity if one writes a proposition without parentheses, it is always recommended to write propositions with parentheses. It is similar to programming languages where writing parentheses is one of the recommended practices in programming. This is purely for better understanding.

2.5 De Morgan’s Laws and other logical equivalences

The simplest format of De Morgan’s law is observed by

\[ \neg(X \land Y) \equiv \neg X \lor \neg Y \]
\[ \neg(X \lor Y) \equiv \neg X \land \neg Y \]

Essentially, when the negation is distributed within the parentheses, \lor is flipped to \land and vice versa. This applies to more general form with multiple propositions within the parentheses as well.

De Morgan’s law shows one type of logical equivalence, and more can be observed from the following tables (captured from page 27-28 of the textbook).

<table>
<thead>
<tr>
<th>TABLE 6 Logical Equivalences.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Equivalence</strong></td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>( p \land T \equiv p )</td>
<td>Identity laws</td>
</tr>
<tr>
<td>( p \lor F \equiv p )</td>
<td>Domination laws</td>
</tr>
<tr>
<td>( p \lor T = T )</td>
<td></td>
</tr>
<tr>
<td>( p \land F = F )</td>
<td></td>
</tr>
<tr>
<td>( p \lor p \equiv p )</td>
<td>Idempotent laws</td>
</tr>
<tr>
<td>( p \land p \equiv p )</td>
<td></td>
</tr>
<tr>
<td>( \neg(\neg p) \equiv p )</td>
<td>Double negation law</td>
</tr>
<tr>
<td>( p \lor q \equiv q \lor p )</td>
<td>Commutative laws</td>
</tr>
<tr>
<td>( p \land q \equiv q \land p )</td>
<td></td>
</tr>
<tr>
<td>( (p \lor q) \land r \equiv p \lor (q \land r) )</td>
<td>Associative laws</td>
</tr>
<tr>
<td>( (p \land q) \lor r \equiv p \land (q \lor r) )</td>
<td></td>
</tr>
<tr>
<td>( p \lor (q \land r) \equiv (p \lor q) \land (p \lor r) )</td>
<td>Distributive laws</td>
</tr>
<tr>
<td>( p \land (q \lor r) \equiv (p \land q) \lor (p \land r) )</td>
<td></td>
</tr>
<tr>
<td>( \neg(p \land q) \equiv \neg p \lor \neg q )</td>
<td>De Morgan’s laws</td>
</tr>
<tr>
<td>( \neg(p \lor q) \equiv \neg p \land \neg q )</td>
<td></td>
</tr>
<tr>
<td>( p \lor (p \land q) \equiv p )</td>
<td>Absorption laws</td>
</tr>
<tr>
<td>( p \land (p \lor q) \equiv p )</td>
<td></td>
</tr>
<tr>
<td>( p \lor \neg p \equiv T )</td>
<td>Negation laws</td>
</tr>
<tr>
<td>( p \land \neg p \equiv F )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE 7 Logical Equivalences Involving Conditional Statements.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p \to q \equiv \neg p \lor q )</td>
</tr>
<tr>
<td>( p \to q \equiv \neg q \to \neg p )</td>
</tr>
<tr>
<td>( p \lor q \equiv \neg p \to q )</td>
</tr>
<tr>
<td>( p \land q \equiv (p \lor q) \to \neg q )</td>
</tr>
<tr>
<td>( \neg(p \to q) \equiv p \land \neg q )</td>
</tr>
<tr>
<td>( (p \to q) \lor (p \to r) \equiv p \to (q \land r) )</td>
</tr>
<tr>
<td>( (p \to r) \land (q \to r) \equiv (p \lor q) \to r )</td>
</tr>
<tr>
<td>( (p \to q) \lor (p \to r) \equiv p \to (q \lor r) )</td>
</tr>
<tr>
<td>( (p \to r) \lor (q \to r) \equiv (p \land q) \to r )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE 8 Logical Equivalences Involving Biconditional Statements.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p \leftrightarrow q \equiv (p \to q) \land (q \to p) )</td>
</tr>
<tr>
<td>( p \leftrightarrow q \equiv \neg p \leftrightarrow \neg q )</td>
</tr>
<tr>
<td>( p \leftrightarrow q \equiv (p \land q) \lor (\neg p \land \neg q) )</td>
</tr>
<tr>
<td>( \neg(p \leftrightarrow q) \equiv p \leftrightarrow \neg q )</td>
</tr>
</tbody>
</table>
2.6 Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF)

Theoretically, by replacing the propositions using the logical equivalences, any proposition can be turned into the form

\[(P_{11} \lor P_{12} \lor P_{13} \lor \cdots) \land (P_{21} \lor P_{22} \lor P_{23} \lor \cdots) \land \cdots = \bigwedge_i \left( \bigvee_j P_{ij} \right)\]

which is the form called Conjunctive Normal Form (CNF), or another form

\[(Q_{11} \land Q_{12} \land Q_{13} \land \cdots) \lor (Q_{21} \land Q_{22} \land Q_{23} \land \cdots) \lor \cdots = \bigvee_i \left( \bigwedge_j P_{ij} \right)\]

which is the form called Disjunctive Normal Form (DNF). CNF implies the terms are connected with a conjunctive operator \(\land\), and DNF implies the terms are connected with a disjunctive operator \(\lor\).

These two forms are used to model the propositions because any arbitrarily complicated proposition can be converted into these two forms.

3 Predicate logic

The propositional logic is powerful but quite limited. Many things in the real life cannot be succinctly expressed with it because propositional logic only deals with propositions and does not extend to objects, their properties and relationship. Predicate logic is an extension of the propositional logic where variables and quantifiers are introduced to be more expressive. More formally, the following components are introduced:

- Variables: \(x, y, z, \cdots\)
- Predicates: \(P(x), Q(x, y, z), \cdots\)
- Quantifiers: \(\forall, \exists, \exists!\)

Let’s see via following example how predicates extend the propositional logic.

\[P(x, y): \text{an integer } x \text{ is greater than } y.\]

This predicate expresses infinitely many propositions by having a variable \(x\). For example, \(P(1, 0)\) is a proposition whose truth value is \(T\) and \(P(-1, 0)\) is another proposition whose truth value is \(F\), and apparently there are infinitely many such propositions. As such, by introducing a variable, we can more easily express the propositions succinctly.

3.1 Quantifiers

Two quantifiers are particularly important in computer science: \(\forall\) and \(\exists\). A variable and a predicate is followed by to describe the quantification of the variable in the predicate. For example

- \(\forall x: P(x)\) refers to ‘\(P(x)\) is true for every \(x\) in the domain \(U\)’
- \(\exists x: P(x)\) refers to ‘\(P(x)\) is true for at least one \(x\) in the domain \(U\)’

We then say the quantifier binds the variable in the predicate, or the variable is bound to the predicate. Often, for the simplicity, it is also written as \(\forall x \in U: P(x)\) and \(\exists x \in U: P(x)\) respectively.
3.1.1 Universal quantifier \( \forall \)

To let \( \forall x : P(x) \) be true, \( P(x) \) has to be true for every \( x \) in the domain. For example,

- \( P(x) \) denotes \( 'x \geq 0' \) and \( x \)'s domain is integers. Then, \( \forall x : P(x) \) is false.
- \( P(x) \) denotes \( 'x \geq 0' \) and \( x \)'s domain is \([5, +\infty)\). Then, \( \forall x : P(x) \) is true.

3.1.2 Existential quantifier \( \exists \)

To let \( \exists x : P(x) \) be true, \( P(x) \) has to be true for at least one \( x \) in the domain. For example,

- \( P(x) \) denotes \( 'x \geq 0' \) and \( x \)'s domain is integers. Then, \( \exists x : P(x) \) is true.
- \( P(x) \) denotes \( 'x \geq 0' \) and \( x \)'s domain is \((-\infty, 0]\). Then, \( \exists x : P(x) \) is false.

\( \exists ! \) is another quantifier that is closely related to \( \exists \). \( \exists ! x : P(x) \) is true if there is exactly one \( x \) in the domain which makes \( P(x) \) true. For example,

- \( P(x) \) denotes \( 'x \geq 0' \) and \( x \)'s domain is integers. Then, \( \exists ! x : P(x) \) is false.
- \( P(x) \) denotes \( 'x \geq 0' \) and \( x \)'s domain is \((-\infty, 0]\). Then, \( \exists ! x : P(x) \) is true.

As we will describe later, \( \exists ! \) can be replaced by a compound predicates of \( \forall \) and \( \exists \).

3.1.3 Priorities of quantifiers

The standard is, for example, \( \forall x : P(x) \lor Q(x) \equiv (\forall x : P(x)) \lor Q(x) \), but many people interpret \( \forall x : P(x) \lor Q(x) \) as \( \forall x : (P(x) \lor Q(x)) \), which is wrong but observed frequently. Therefore, to avoid ambiguity, it is also recommended to add parentheses.

3.1.4 De Morgan’s laws in predicates

Application of De Morgan’s laws in the predicates is similar to that in the propositions. Besides flipping between \( \land \) and \( \lor \), we further flip \( \exists \) to \( \forall \) and \( \forall \) to \( \exists \). For example, in the domain \( U \),

\[ \neg (\forall x : P(x)) \equiv \exists x : \neg P(x) \]

As we mentioned earlier, it is often abbreviated as \( \forall x \in U : P(x) \). This implies,

\[ \neg (\forall x \in U : P(x)) \equiv \exists x \in U : \neg P(x) \] instead of \( \exists x \notin U : \neg P(x) \)

because \( '\in U' \) is not an operator nor a proposition in \( \forall x \in U : P(x) \).

3.2 Translation from English to predicates

It is often important to translate the real-life phenomena to predicates. Let’s try some examples of correct translation.

1. Every student in CS330 is a student of CS department. Domain \( U \) is all students in CS330 and \( P(x) \) denotes ‘the student \( x \) is in CS’.
• \(\forall x \in U : P(x)\)

2. Every student in CS330 is a student of CS department. Domain \(U\) is all students in IIT, \(P(x)\) denotes 'the student \(x\) is in CS', and \(Q(x)\) denotes 'the student is taking CS330'.

• \(\forall x \in U : Q(x) \rightarrow P(x)\)

3. There is at least one CS student in CS330 this semester. Domain \(U\) is all students in CS330 and \(P(x)\) denotes 'the student \(x\) is in CS'.

• \(\exists x \in U : P(x)\)

4. There is at least one CS student in CS330 this semester. Domain \(U\) is all students in IIT, \(P(x)\) denotes 'the student \(x\) is in CS', and \(Q(x)\) denotes 'the student is taking CS330'.

• \(\exists x \in U : Q(x) \land P(x)\)

The first and the third ones are easy to understand, but the other two are not. Why can’t we have \(\forall x \in U : Q(x) \land P(x)\) or \(\exists x \in U : Q(x) \rightarrow P(x)\)?

• \(\forall x \in U : Q(x) \land P(x)\): This means every student in IIT must be a student of computer science department and every student in IIT must take CS330, which is quite different from the original sentence in 2.

• \(\exists x \in U : Q(x) \rightarrow P(x)\): This means there exists at least one student \(x\) where \(Q(x) \rightarrow P(x)\) is true. Recall that \(F \rightarrow F \equiv T\), therefore this predicate is always true regardless of whether there is a CS student in CS330 because there exists at least one \(x\) in IIT where \(Q(x) = F\). Therefore, what this predicate means is also quite different from the original sentence in 4.

A rule of thumb is that, \(\rightarrow\) appears after \(\forall\), and \(\land\) appears after \(\exists\). However this belongs to heuristics only, and it does not always hold.

### 3.3 Nested quantifiers

The real life is much more complicated, and one quantifier is often not enough to express things in our life. Therefore, multiple quantifiers are nested to express more complicated statements.

• \(\forall x \forall y : P(x,y)\): For every \(x\) and \(y\) in the corresponding domains, \(P(x,y)\) holds.

• \(\forall x \exists y : P(x,y)\): For every \(x\) in the domain, there is at least one \(y\) which makes \(P(x,y)\) true.

• \(\exists x \forall y : P(x,y)\): There is at least one \(x\) in the domain where every \(y\) in the domain makes \(P(x,y)\) true.

• \(\exists x \exists y : P(x,y)\): There exists at least one pair \((x,y)\) in the corresponding domains which makes \(P(x,y)\) true.

Let’s see by several examples what are the differences.

- \(P(x,y) : xy = 0\), domain of \(x, y\) are real numbers.
  - \(\forall x \forall y : P(x,y)\): False. If \(x \neq 0\), \(P(x,y)\) does not hold for every \(y\).
  - \(\forall x \exists y : P(x,y)\): True. For any \(x, y = 0\) makes \(P(x,y)\) true.
  - \(\exists x \forall y : P(x,y)\): True. For \(x = 0\), any \(y\) makes \(P(x,y)\) true.
– \( \exists x \exists y : P(x, y) \): True. For \( x = 0, y = 23 \), \( P(x, y) \) is true. (This is not the only pair though)

- \( P(x, y) : \frac{x}{y} = 1 \), domain of \( x, y \) are real numbers.
  - \( \forall x \forall y : P(x, y) \): False. \( P(x, y) \) does not hold for any pair of \( x, y \).
  - \( \forall x \exists y : P(x, y) \): True. For any \( x, y = x \) makes \( P(x, y) \) true.
  - \( \exists x \forall y : P(x, y) \): False. An \( x \) where \( P(x, y) \) is true for every \( y \) in the domain does not exist.
  - \( \exists x \exists y : P(x, y) \): True. For \( x = 3, y = 3 \), \( P(x, y) \) is true.

* What will be enough to disprove the following predicates in general? +2

- \( \forall x \forall y : Q(x, y) \)
- \( \forall x \exists y : Q(x, y) \)
- \( \exists x \forall y : Q(x, y) \)
- \( \exists x \exists y : Q(x, y) \)

* Can you write an equivalent predicate of \( \exists ! P(x) \) with \( \forall \) and \( \exists \) only? +4

4 Proof and disproof

Proofs have many practical applications in cyber physical systems.

- Verification that computer programs are correct.
- Proving systems are secure.
- Inferences in artificial intelligence.
- Showing the consistencies in system specifications.

Especially, the mathematical proof is the core of this course, and this course focuses on the following two types.

1. Proof by contradiction: assume the statement is wrong and derive a contradiction.

Those two types are of particular interest in computer science.

4.1 Basic principal of proof

To prove a statement is correct, one needs to show that the statement is always true under any circumstance defined in the statement (e.g., for any element in the domain). Showing one case where the statement holds does not qualify as a correct proof. This makes the mathematical proof quite different from common sense or belief in our real life. For example,

- Apple computers are more expensive than Samsung laptops: this is not always true, therefore it is false.
\begin{itemize}
  \item Mexican food is spicy: not all Mexican food is spicy, therefore it is false.
  \item Living cost in California is higher than in Illinois: it depends on the regions in the states, therefore it is false.
\end{itemize}

Therefore, you have to argue the statement holds in every single case in order to say it is true, and if you can find out one single case where the statement does not hold, you can conclude the statement is false.

### 4.2 Proof by contradiction

The essence of this technique is that to prove \( A \Rightarrow B \), we will instead show \( \neg B \Rightarrow \neg A \). As we mentioned earlier, it can be shown that these two forms are logically equivalent to each other because one is a contrapositive of the other.

#### 4.2.1 A sample proof by contradiction

**Theorem:** \( \sqrt{2} \) is irrational. That is, \( \sqrt{2} \) cannot be written as \( \frac{a}{b} \), where \( a \) and \( b \) are integers with no common factors.

We first need to convert this to the \( A \Rightarrow B \) form as above. One simple conversion is \( T \Rightarrow \sqrt{2} \) is irrational. (Convince yourself by examining a truth table that this is indeed a valid conversion.)

**Proof by contradiction:** We will show that \( \sqrt{2} \) is rational implies \( F \) — that is, that if we assume that \( \sqrt{2} \) is rational, we can derive a contradiction.

By the definition of rationality, \( \sqrt{2} = \frac{a}{b} \), for two relatively prime integers \( a \) and \( b \). Thus \( \sqrt{2} \cdot b = a \). It follows that \( 2b^2 = a^2 \) and, by the definition of an even number, that \( a^2 \) is even.

We now take a small diversion to help us arrive at our goal.

**Lemma:** \( a^2 \) is even \( \Rightarrow \) \( a \) is even

**Proof by contradiction:** We show \( a \) is odd \( \Rightarrow \) \( a^2 \) is odd. Since \( a \) is odd, it has the form \( 2n + 1 \), for some integer \( n \). Thus \( a^2 = (2n + 1)^2 = 4n^2 + 4n + 1 = 2(2n^2 + 2n) + 1 \). Thus \( a^2 \) is odd.

Now that we have concluded that \( a^2 \) is even \( \Rightarrow \) \( a \) is even, we can resume our original proof. Since \( a \) is even, it can be written as \( 2c \), where \( c \) is an integer. Thus \( 2b^2 = (2c)^2 \), so \( 2b^2 = 4c^2 \), and \( b^2 = 2c^2 \). Thus \( b^2 \) is even, and by the lemma we know that \( b \) is even.

So both \( a \) and \( b \) are even. They share the common factor 2. But we originally assumed that \( a \) and \( b \) had no common factors! Thus we have arrived at a contradiction and proved the original theorem, that \( \sqrt{2} \) is irrational.

### 4.3 Proof by induction

Mathematical induction is a quite powerful tool, which belongs to a special case of direct proof. It is particularly useful in proving statements in infinite domains. In some cases, it is easy to show the relationship between a case and a case ‘next’ to it, and the mathematical induction generalizes such relationship to prove that the proved phenomena can be observed in the entire domain. It must contain the following components.

- **Base case:** Show the statement holds for a case with small size (usually the smallest number in the domain).
- **Inductive hypothesis:** Assume the statement holds for a case with arbitrary size \( k \).
- **Inductive step:** Show that the statement also holds for a case with the size \( k + 1 \).
4.3.1 A sample proof by induction

Let \( P(n) \) denote a statement described with a positive integer \( n \), e.g., \( '1 + 3 + 5 + \cdots + 2n - 1 = n^2' \), and let's say we are asked to prove the equation holds for every positive integer by induction. Then, our proof proceeds as follows.

- **Base case:** The smallest number in the domain is 1. In this case, LHS is 1, and RHS is \( 1^2 = 1 \) as well, therefore the equation holds.

- **Inductive hypothesis:** We assume \( P(k) \) holds for an arbitrary integer \( k \). That is, \( 1+3+5+\cdots+2k-1 = k^2 \).

- **Inductive step:** We need to show \( P(k+1) \) also holds.

The hardest part in the proof is the inductive step. There is no systematized way to complete the inductive step, but one of the heuristics is to write down the statement with the case of \( k+1 \) and then try to identify the case of \( k \) from it. In our previous example, we have

\[
P(k+1) : 1 + 3 + 5 + \cdots + 2k - 1 + 2k + 1 = (k + 1)^2
\]

Because we have assumed \( P(k) \) holds for \( k \), we know \( 1 + 3 + 5 + \cdots + 2k - 1 \) can be substituted with \( k^2 \). Then, LHS of \( P(k+1) \) is equal to

\[
\text{LHS of } P(k+1) = k^2 + 2k + 1
\]

Then, we also know

\[
\text{RHS of } P(k+1) = (k + 1)^2 = k^2 + 2k + 1
\]

Since LHS=RHS in \( P(k+1) \), we can conclude that \( P(k+1) \) is true if \( P(k) \) is also true.

The reason proof by induction works

Essentially, we first show the base case \( P(1) \) is true by showing that special case directly. Since the base case is often an example of small size, it is always possible to manually prove the statement holds in that example. What we do subsequently is to show \( P(k) \to P(k+1) \) is true for any integer \( k \), and this applies to 1 as well. Therefore, we have \( P(1) \to P(2) \), and \( P(2) \to P(3) \) can be shown as well because we showed that the inductive step works for any integer \( k \). Repeatedly, we have \( P(3) \to P(4) \), and this will go on infinitely one by one. Because we just showed in the base case that \( P(1) = T \), the only possible truth value of \( P(2) = T \), and the above ‘chain reaction’ will show that \( P(k) = T \) for every integer \( k \geq 1 \).

The other format of induction

In some other cases, one needs to prove

- **Inequality:** \( P(n) \) is an inequality about \( n \), e.g., \( P(n) : 2^n < n! \)

- **Special structure:** \( P(n) \) is a statement concerning a structure of size \( n \), e.g., about a tree of height \( n \), a graph of \( n \) vertices.

They are still proved with the same components, but it is much harder to use the previous heuristics to show the inductive step. Unfortunately, there is no systematized way to do such proof in those cases, and we need experiences to ‘guess’ what will be a right direction to follow in the proof. After then, we need to try different logical deductions until we find out the correct one.
4.4 Disproof

In the previous two sections, we were trying to prove that the statement is always true. What if we want to argue that the statement is wrong? Such a process is called disproof, and it is (usually) much easier than the proof.

Essentially, if one wants to disprove $P(x)$, he only needs to show that ‘$P(x)$ is always true’ is wrong. That is, we need to prove:

$$\neg(\forall x : P(x)) \equiv \exists x : \neg P(x)$$

Thus, we only need to find out one $x$ in the domain which shows that $P(x)$ is false. Such a case or example is called counterexample, and finding out a counterexample suffices to disprove a statement. For example,

- Disprove that $x^2 > 0$ in the domain of integers: 0 is a counterexample. Square of it is not positive.
- Disprove that Apple products are more expensive than Samsung products: iPhone SE is cheaper than Galaxy S7 edge.