Lesson 0. Preliminary

CSIE 3110 – Formal Languages and Automata Theory

Tony Tan
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Table of contents

1. Introduction
2. Some words about mathematical proofs
3. The halting problem in C++
4. The notion of alphabets and languages
5. Concluding remarks

Table of contents

1. Introduction

2. Some words about mathematical proofs

3. The halting problem in C++

4. The notion of alphabets and languages

5. Concluding remarks

The most important information!

Official course website:

`https://www.csie.ntu.edu.tw/~tonytan/teaching/2021a-aut/2021a-aut.html`

The most important information!

Official course website:

`https://www.csie.ntu.edu.tw/~tonytan/teaching/2021a-aut/2021a-aut.html`

- All information about this course can be found in the website.

The most important information!

Official course website:

`https://www.csie.ntu.edu.tw/~tonytan/teaching/2021a-aut/2021a-aut.html`

- All information about this course can be found in the website.
- Lecture notes, slides and homework will be posted there.

The most important information!

Official course website:

`https://www.csie.ntu.edu.tw/~tonytan/teaching/2021a-aut/2021a-aut.html`

- All information about this course can be found in the website.
- Lecture notes, slides and homework will be posted there.
- Information about the midterm and final exams will be posted there.

The most important information!

Official course website:

`https://www.csie.ntu.edu.tw/~tonytan/teaching/2021a-aut/2021a-aut.html`

- All information about this course can be found in the website.
- Lecture notes, slides and homework will be posted there.
- Information about the midterm and final exams will be posted there.
- Videos of lectures will be posted in NTU COOL.

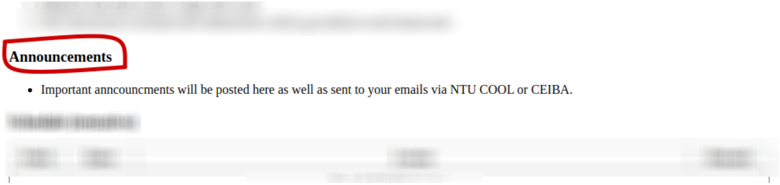
The most important information!

Official course website:

`https://www.csie.ntu.edu.tw/~tonytan/teaching/2021a-aut/2021a-aut.html`

- All information about this course can be found in the website.
- Lecture notes, slides and homework will be posted there.
- Information about the midterm and final exams will be posted there.
- Videos of lectures will be posted in NTU COOL.

Pay special attention to the "Announcement" part:



Announcements

- Important anncouncments will be posted here as well as sent to your emails via NTU COOL or CEIBA.

Staff

Instructor:

- Tony Tan (陳偉松)
  `tonytan@csie.ntu.edu.tw`

TA:

- Lu Chia-Hsuan (呂佳軒)
  `r09922064@csie.ntu.edu.tw`
- Lu Yu-Cheng (呂侑承)
  `r109220304@csie.ntu.edu.tw`
- Mailing list: `automata@csie.ntu.edu.tw`.

Syllabus and schedule (can be found in the course website)

Week	Dates	Lesson	Remark
Part 0: Preliminaries			
I	27 Sep.	Lesson 0. Preliminaries	--
Part 1: Regular languages and context-free languages			
II	4 Oct.	Lesson 1. Finite state automata	--
III	11 Oct.	**Holiday (National day)**	--
IV	18 Oct.	Lesson 2. Regular expressions	--
V	25 Oct.	Lesson 3. Context-free languages	--
VI	1 Nov.	Lesson 4. Push-down automata	--
Reading week and midterm exam			
VII	8 Nov.	Reading week	--
VIII	15 Nov.	**Midterm exam:**Monday, 10:30-12:30, room: To be determined.	--
Part 2: Turing machines and some basic complexity classes			
IX	22 Nov.	Lesson 5. Turing machines and decidable languages	--
X	29 Nov.	Lesson 6. Turing machines and the notion of algorithm	--
XI	6 Dec.	Lesson 7. Universal Turing machines and halting problem	--
XII	13 Dec.	Lesson 8. Reducibility	--
XIII	20 Dec.	Lesson 9. Non-deterministic Turing machines	--
XIV	27 Dec.	Lesson 10. Basic complexity classes	--
Reading week and final exam			
XV	3 Jan.	Reading week	--
XVI	10 Jan.	**Final exam:** Monday, 10:30--12:30, room: To be determined.	--

How this course will be conducted

- The note for a lesson in a particular week will be posted in the course website a week before.

II	4 Oct.	Lesson 1. Finite state automata (tex, pdf, slides)

For example, for lesson 1 it will be posted 1 week before 4 October.

How this course will be conducted

- The note for a lesson in a particular week will be posted in the course website a week before.

II	4 Oct.	Lesson 1. Finite state automata (tex, pdf, slides)

  For example, for lesson 1 it will be posted 1 week before 4 October.

- Weekly online discussion on Monday, starting at 10:30 am.

  For now, we use Google meet.
  Check the announcement "Our Google meet link" in NTU COOL for the link.

How this course will be conducted

- The note for a lesson in a particular week will be posted in the course website a week before.

[illegible]	[illegible]	[illegible]
II	4 Oct.	Lesson 1. Finite state automata (tex, pdf, slides)
[illegible]	[illegible]	[illegible]

  For example, for lesson 1 it will be posted 1 week before 4 October.

- Weekly online discussion on Monday, starting at 10:30 am.

  For now, we use Google meet.
  Check the announcement "Our Google meet link" in NTU COOL for the link.

- Depending on the situation, we may be able to conduct the lesson in the physical class, but nothing is certain yet.

  It all depends on the future instruction from the university.

About the weekly discussion

About the weekly discussion

- The purpose of the online discussion is for you to ask questions.

About the weekly discussion

- The purpose of the online discussion is for you to ask questions.
- You can ask any questions, and I will try my best to answer.

About the weekly discussion

- The purpose of the online discussion is for you to ask questions.
- You can ask any questions, and I will try my best to answer.
- Depending on the questions, I will even explain materials already covered in the video.

  I will not repeat the whole lecture during the discussion.

About the weekly discussion

- The purpose of the online discussion is for you to ask questions.
- You can ask any questions, and I will try my best to answer.
- Depending on the questions, I will even explain materials already covered in the video.

  I will not repeat the whole lecture during the discussion.
- Since this course is supposed to be in English, the discussion is also in English.

Some details about videos

Usually each lesson will be divided into a few videos.

Some details about videos

Usually each lesson will be divided into a few videos.

For example, for lesson 1:

II	4 Oct.	Lesson 1. Finite state automata (tex, pdf, slides)

There will be three videos, one video for each of the following topics.

Some details about videos

Usually each lesson will be divided into a few videos.

For example, for lesson 1:

II	4 Oct.	Lesson 1. Finite state automata (tex, pdf, slides)

There will be three videos, one video for each of the following topics.

- Deterministic finite state automata.
- Non-deterministic finite state automata.
- Pumping lemma.

Textbook, homework and exams

Textbook, homework and exams

- Textbook: *Introduction to the Theory of Computation* by M. Sipser.

  But we will not follow the book strictly.

Textbook, homework and exams

- Textbook: *Introduction to the Theory of Computation* by M. Sipser.

  But we will not follow the book strictly.

- There will be two homework.

  Each weighs 20%.

Textbook, homework and exams

- Textbook: *Introduction to the Theory of Computation* by M. Sipser.

  But we will not follow the book strictly.

- There will be two homework.

  Each weighs 20%.

- There will be one midterm exam and one final exam.

  Each weighs 30%.

Textbook, homework and exams

- Textbook: *Introduction to the Theory of Computation* by M. Sipser.

  But we will not follow the book strictly.

- There will be two homework.

  Each weighs 20%.

- There will be one midterm exam and one final exam.

  Each weighs 30%.

- The instruction on how to submit your homework will be provided in due time.

Textbook, homework and exams

- Textbook: *Introduction to the Theory of Computation* by M. Sipser.

  But we will not follow the book strictly.

- There will be two homework.

  Each weighs 20%.

- There will be one midterm exam and one final exam.

  Each weighs 30%.

- The instruction on how to submit your homework will be provided in due time.

- We are still deciding how to conduct the exams.

Table of contents

1. Introduction

2. Some words about mathematical proofs

3. The halting problem in C++

4. The notion of alphabets and languages

5. Concluding remarks

Definitions, theorems, proofs[1]

[1] Adopted from Chapter 0 in Sipser's textbook

Definitions, theorems, proofs[1]

Definitions describe the objects and notions that we use.

[1] Adopted from Chapter 0 in Sipser's textbook

Definitions, theorems, proofs[1]

Definitions describe the objects and notions that we use.

Precision is essential to any mathematical definition.

[1] Adopted from Chapter 0 in Sipser's textbook

Definitions, theorems, proofs[1]

Definitions describe the objects and notions that we use.

Precision is essential to any mathematical definition.

When defining some object, we must make clear what constitutes that object and what does not.

[1] Adopted from Chapter 0 in Sipser's textbook

Definitions, theorems, proofs[1]

Definitions describe the objects and notions that we use.

Precision is essential to any mathematical definition.

When defining some object, we must make clear what constitutes that object and what does not.

A **theorem** is a mathematical statement proved true.

[1] Adopted from Chapter 0 in Sipser's textbook

Definitions, theorems, proofs[1]

Definitions describe the objects and notions that we use.

Precision is essential to any mathematical definition.

When defining some object, we must make clear what constitutes that object and what does not.

A **theorem** is a mathematical statement proved true.

Generally we reserve the use of that word for statements of special interest.

[1] Adopted from Chapter 0 in Sipser's textbook

Definitions, theorems, proofs[1]

Definitions describe the objects and notions that we use.

Precision is essential to any mathematical definition.

When defining some object, we must make clear what constitutes that object and what does not.

A **theorem** is a mathematical statement proved true.

Generally we reserve the use of that word for statements of special interest.

A **lemma** is a statement that we prove to assist in the proof of a theorem or another lemma.

[1] Adopted from Chapter 0 in Sipser's textbook

Definitions, theorems, proofs[1]

Definitions describe the objects and notions that we use.

Precision is essential to any mathematical definition.

When defining some object, we must make clear what constitutes that object and what does not.

A **theorem** is a mathematical statement proved true.

Generally we reserve the use of that word for statements of special interest.

A **lemma** is a statement that we prove to assist in the proof of a theorem or another lemma.

A **corollary** (of a theorem) is a statement that follows easily from a theorem or its proof.

[1] Adopted from Chapter 0 in Sipser's textbook

Mathematical proofs

Mathematical proofs

A **mathematical proof** (or, in short **proof**) is a “convincing” logical argument that a statement is true.

Mathematical proofs

A **mathematical proof** (or, in short **proof**) is a "convincing" logical argument that a statement is true.

Formally we can view a proof as a (finite) sequence of statements:

$$\text{statement}_1$$

$$\text{statement}_2$$

$$\vdots$$

$$\text{statement}_n$$

such that each statement s_i is either an assumption or it can be trivially deduced from earlier statements $s_1, \ldots, s_{i-1}$.

Mathematical proofs

A **mathematical proof** (or, in short **proof**) is a "convincing" logical argument that a statement is true.

Formally we can view a proof as a (finite) sequence of statements:

$$\text{statement}_1$$
$$\text{statement}_2$$
$$\vdots$$
$$\text{statement}_n$$

such that each statement s_i is either an assumption or it can be trivially deduced from earlier statements $s_1, \ldots, s_{i-1}$.

statement_n is the theorem/lemma that we want to prove.

Some examples of simple deductions

Example 1:

$$\begin{array}{l} \text{If it is raining, John stays at home.} \\ \text{John is not at home today.} \\ \hline \therefore \text{ It is not raining today.} \end{array}$$

Some examples of simple deductions

Example 1:

If it is raining, John stays at home.
John is not at home today.

$\therefore$ It is not raining today.

Example 2:

If it is sunny, John goes to the beach.
When John is at the beach, he swims in the sea.
It is sunny today.

$\therefore$ Today John swims in the sea.

Some examples of simple deductions

Example 1:

If it is raining, John stays at home.
John is not at home today.

$\therefore$ It is not raining today.

Example 2:

If it is sunny, John goes to the beach.
When John is at the beach, he swims in the sea.
It is sunny today.

$\therefore$ Today John swims in the sea.

See Appendix B for some other examples of deductions.

Types of proofs

[2] For more details and examples, see Chapter 0 in Sipser's textbook.

Types of proofs

Types of proofs that normally occur in this course.[2]

- Proofs by construction.
- Proofs by contradiction.
- Proofs by induction.

[2] For more details and examples, see Chapter 0 in Sipser's textbook.

Tips to write proofs[3]

[3] Taken from Chapter 0 in Sipser's textbook.

Tips to write proofs[3]

Be patient. If you don't see how to do it right away, don't worry. Researchers sometimes work for weeks or even years to find a single proof.

[3] Taken from Chapter 0 in Sipser's textbook.

Tips to write proofs[3]

Be patient. If you don't see how to do it right away, don't worry. Researchers sometimes work for weeks or even years to find a single proof.

Genius is the patience of thoughts, concentrated in a certain direction.

— Isaac Newton

[3]Taken from Chapter 0 in Sipser's textbook.

Tips to write proofs[3]

Be patient. If you don't see how to do it right away, don't worry. Researchers sometimes work for weeks or even years to find a single proof.

> *Genius is the patience of thoughts, concentrated in a certain direction.*
>
> — *Isaac Newton*

Come back to it. Look over the statement you want to prove, think about it a bit, leave it, and then return a few minutes or hours later. Let the unconscious, intuitive part of your mind have a chance to work.

[3] Taken from Chapter 0 in Sipser's textbook.

Tips to write proofs[4]

Be neat! When you are building your intuition for the statement you are trying to prove, use simple, clear pictures and/or text. You are trying to develop your insight into the statement, and sloppiness gets in the way of insight. Furthermore, when you are writing a solution for another person to read, neatness will help that person understand it.

[4] Taken from Chapter 0 in Sipser's textbook.

Tips to write proofs[4]

Be neat! When you are building your intuition for the statement you are trying to prove, use simple, clear pictures and/or text. You are trying to develop your insight into the statement, and sloppiness gets in the way of insight. Furthermore, when you are writing a solution for another person to read, neatness will help that person understand it.

Be concise. Brevity helps you express high-level ideas without getting lost in details. Good mathematical notation is useful for expressing ideas concisely. But be sure to include enough of your reasoning when writing up a proof so that the reader can easily understand what you are trying to say

[4] Taken from Chapter 0 in Sipser's textbook.

Tips to write proofs[5]

Be patient.

Come back to it.

Be neat!

Be concise.

[5] Taken from Chapter 0 in Sipser's textbook.

After you have written down your proof....[6]

Reread, reconsider and reexamine your proof, even after you are convinced that your proof is correct.

[6] Adopted from the book *How to solve it* by George Pólya.

After you have written down your proof....[6]

Reread, reconsider and reexamine your proof, even after you are convinced that your proof is correct.

Doing so could consolidate your knowledge and develop your problem solving skill.

[6] Adopted from the book *How to solve it* by George Pólya.

After you have written down your proof....[6]

Reread, reconsider and reexamine your proof, even after you are convinced that your proof is correct.

Doing so could consolidate your knowledge and develop your problem solving skill.

Ask the following questions.

- What is the main idea of the proof?
- Can the proof be simplified?
- Can the result be derived differently?
- Can the result/method be used for some other problem?
- How does it relate to other results that you know?

All these obviously take time and energy, but worth the effort.

[6] Adopted from the book *How to solve it* by George Pólya.

Some references

On general problem solving skill:

- *How to Solve it* by George Pólya.
- *Princeton Companion to Mathematics*, part VIII. *Final Perspectives*, Timothy Gowers, editor.

On the more technical side:

- *Mathematical discovery on understanding, learning, and teaching problem solving* (volumes I and II) by George Pólya.
- *Solving mathematical problems: A personal perspective* by Terrence Tao.

Table of contents

1. Introduction

2. Some words about mathematical proofs

3. The halting problem in C++

4. The notion of alphabets and languages

5. Concluding remarks

An example of impossible problem for computer

An example of impossible problem for computer

Consider the following problem denoted by **Problem-A**.

Problem-A

Input: Two files:
The first file is a C++ program, denoted by `file-1.cpp`.
The second file is a file with arbitrary extension, denoted by `file-2`.

Task: Output `True`, if the C++ program file-1.cpp returns `True`
when the input (to `file-1.cpp`) is the content of `file-2`.
Otherwise, output `False`.

An example of impossible problem for computer

Consider the following problem denoted by **Problem-A**.

Problem-A

Input: Two files:
The first file is a C++ program, denoted by `file-1.cpp`.
The second file is a file with arbitrary extension, denoted by `file-2`.

Task: Output `True`, if the C++ program file-1.cpp returns `True`
when the input (to `file-1.cpp`) is the content of `file-2`.
Otherwise, output `False`.

(Important!) Notice how we define **Problem-A**.

It is clear what the input and output should be!

An example of impossible problem for computer

Consider the following problem denoted by **Problem-A**.

Problem-A

Input: Two files:
The first file is a C++ program, denoted by `file-1.cpp`.
The second file is a file with arbitrary extension, denoted by `file-2`.

Task: Output `True`, if the C++ program file-1.cpp returns `True`
when the input (to `file-1.cpp`) is the content of `file-2`.
Otherwise, output `False`.

(Important!) Notice how we define **Problem-A**.

It is clear what the input and output should be!

This is not acceptable:

- We want to decide if a C++ program output `True` on an input.
- We want to decide the outcome of a C++ program on an input.
- **Problem-A** is to determine the outcome of a C++ program on an input.

A little analysis: What's wrong with these specifications?

- We want to decide if a C++ program output `True` on an input.
- We want to decide the outcome of a C++ program on an input.
- **Problem-A** is to determine the outcome of a C++ program on an input.

A little analysis: What's wrong with these specifications?

- We want to decide if a C++ program output `True` on an input.
- We want to decide the outcome of a C++ program on an input.
- **Problem-A** is to determine the outcome of a C++ program on an input.

It is not clear if the C++ program or the input is fixed or both are fixed.

A little analysis: What's wrong with these specifications?

- We want to decide if a C++ program output `True` on an input.
- We want to decide the outcome of a C++ program on an input.
- **Problem-A** is to determine the outcome of a C++ program on an input.

It is not clear if the C++ program or the input is fixed or both are fixed.

Problem-X(`P.cpp`)

Input: A file denoted by `input-file`.
Task: Output `True`, if the C++ program `P.cpp` returns `True` on input `input-file`. Otherwise, output `False`.

A little analysis: What's wrong with these specifications?

- We want to decide if a C++ program output `True` on an input.
- We want to decide the outcome of a C++ program on an input.
- **Problem-A** is to determine the outcome of a C++ program on an input.

It is not clear if the C++ program or the input is fixed or both are fixed.

Problem-X(`P.cpp`)

Input: A file denoted by `input-file`.
Task: Output `True`, if the C++ program `P.cpp` returns `True` on input `input-file`. Otherwise, output `False`.

Problem-Y(`input-file`)

Input: A C++ program denoted by `P.cpp`.
Task: Output `True`, if the C++ program `P.cpp` returns `True` on input `input-file`. Otherwise, output `False`.

A little analysis: What's wrong with these specifications?

- We want to decide if a C++ program output `True` on an input.
- We want to decide the outcome of a C++ program on an input.
- **Problem-A** is to determine the outcome of a C++ program on an input.

It is not clear if the C++ program or the input is fixed or both are fixed.

Problem-X(`P.cpp`)

Input: A file denoted by `input-file`.
Task: Output `True`, if the C++ program `P.cpp` returns `True` on input `input-file`. Otherwise, output `False`.

Problem-Y(`input-file`)

Input: A C++ program denoted by `P.cpp`.
Task: Output `True`, if the C++ program `P.cpp` returns `True` on input `input-file`. Otherwise, output `False`.

Problem-Z(`P.cpp`, `input-file`)

Input: Nothing.
Task: Output `True`, if the C++ program `P.cpp` returns `True` on input `input-file`. Otherwise, output `False`.

A little analysis: What's wrong with these specifications?

- We want to decide if a C++ program output `True` on an input.
- We want to decide the outcome of a C++ program on an input.
- **Problem-A** is to determine the outcome of a C++ program on an input.

It is not clear if the C++ program or the input is fixed or both are fixed.

Problem-X(`P.cpp`)

Input: A file denoted by `input-file`.
Task: Output `True`, if the C++ program `P.cpp` returns `True` on input `input-file`. Otherwise, output `False`.

Problem-Y(`input-file`)

Input: A C++ program denoted by `P.cpp`.
Task: Output `True`, if the C++ program `P.cpp` returns `True` on input `input-file`. Otherwise, output `False`.

Problem-Z(`P.cpp`, `input-file`)

Input: Nothing.
Task: Output `True`, if the C++ program `P.cpp` returns `True` on input `input-file`. Otherwise, output `False`.

All of them are very different from our original **Problem-A**.

First principle

To define a problem, write down precisely what the input and output should be.

First principle

To define a problem, write down precisely what the input and output should be.

Before we start designing an algorithm/program/Turing machines, write down precisely what the input and output should be.

First principle

To define a problem, write down precisely what the input and output should be.

<u>Before</u> we start designing an algorithm/program/Turing machines, write down precisely what the input and output should be.

For example, we can use this format:

Problem XYZ	
Input:	...
Task:	...

and

Algorithm XYZ	
Input:	...
Task:	...

Other format is also acceptable as long as the input and output is clear.

Coming back to Problem-A

Problem-A

Input: Two files:
The first file is a C++ program, denoted by `file-1.cpp`.
The second file is a file with arbitrary extension, denoted by `file-2`.

Task: Output `True`, if the C++ program file-1.cpp returns `True`
when the input (to `file-1.cpp`) is the content of `file-2`.
Otherwise, output `False`.

Coming back to Problem-A

Problem-A

Input: Two files:
The first file is a C++ program, denoted by `file-1.cpp`.
The second file is a file with arbitrary extension, denoted by `file-2`.

Task: Output `True`, if the C++ program file-1.cpp returns `True`
when the input (to `file-1.cpp`) is the content of `file-2`.
Otherwise, output `False`.

Note that it requires that on <u>every</u> input `file-1.cpp` and `file-2`, it should output `True` or `False`.

Coming back to Problem-A

Problem-A

Input: Two files:
The first file is a C++ program, denoted by `file-1.cpp`.
The second file is a file with arbitrary extension, denoted by `file-2`.

Task: Output `True`, if the C++ program file-1.cpp returns `True`
when the input (to `file-1.cpp`) is the content of `file-2`.
Otherwise, output `False`.

Note that it requires that on every input `file-1.cpp` and `file-2`, it should output `True` or `False`.

We would like to show:

Theorem 0.1

There is no C++ program for **Problem-A***.*

In other words, it is impossible to write a C++ program for **Problem-A**.

Reductions....

Problem-A

Input: Two files:
The first file is a C++ program, denoted by `file-1.cpp`.
The second file is a file with arbitrary extension, denoted by `file-2`.

Task: Output `True`, if the C++ program file-1.cpp returns `True`
when the input (to `file-1.cpp`) is the content of `file-2`.
Otherwise, output `False`.

We would like to show that there is no C++ program for **Problem-A**, but we don't know how to show it.

Reductions....

Problem-A

Input: Two files:
The first file is a C++ program, denoted by `file-1.cpp`.
The second file is a file with arbitrary extension, denoted by `file-2`.

Task: Output `True`, if the C++ program file-1.cpp returns `True` when the input (to `file-1.cpp`) is the content of `file-2`.
Otherwise, output `False`.

We would like to show that there is no C++ program for **Problem-A**, but we don't know how to show it.

So, we reduce it to:

Problem-B

Input: One files: A C++ program, denoted by `file.cpp`.

Task: Output `True`, if the C++ program file.cpp returns `True` when its input is the content of `file.cpp` itself.
Otherwise, output `False`.

Reductions....

Problem-A

Input: Two files:
The first file is a C++ program, denoted by `file-1.cpp`.
The second file is a file with arbitrary extension, denoted by `file-2`.

Task: Output `True`, if the C++ program file-1.cpp returns `True`
when the input (to `file-1.cpp`) is the content of `file-2`.
Otherwise, output `False`.

We would like to show that there is no C++ program for **Problem-A**, but we don't know how to show it.

So, we reduce it to:

Problem-B

Input: One files: A C++ program, denoted by `file.cpp`.

Task: Output `True`, if the C++ program file.cpp returns `True`
when its input is the content of `file.cpp` itself.
Otherwise, output `False`.

Note that if there is a C++ program for **Problem-A**, then there is a C++ program for **Problem-B**.

Reductions....

> **Problem-A**
>
> **Input:** Two files:
> The first file is a C++ program, denoted by `file-1.cpp`.
> The second file is a file with arbitrary extension, denoted by `file-2`.
>
> **Task:** Output `True`, if the C++ program file-1.cpp returns `True`
> when the input (to `file-1.cpp`) is the content of `file-2`.
> Otherwise, output `False`.

We would like to show that there is no C++ program for **Problem-A**, but we don't know how to show it.

So, we reduce it to:

> **Problem-B**
>
> **Input:** One files: A C++ program, denoted by `file.cpp`.
>
> **Task:** Output `True`, if the C++ program file.cpp returns `True`
> when its input is the content of `file.cpp` itself.
> Otherwise, output `False`.

Note that if there is a C++ program for **Problem-A**, then there is a C++ program for **Problem-B**.

In some sense, **Problem-A** is more "general" than **Problem-B**.

Reductions....

Problem-B

Input: One files: A C++ program, denoted by `file.cpp`.

Task: Output <u>`True`</u>, if the C++ program file.cpp returns `True` when its input is the content of `file.cpp` itself.

Otherwise, output <u>`False`</u>.

Reductions....

Problem-B

Input: One files: A C++ program, denoted by `file.cpp`.

Task: Output <u>`True`</u>, if the C++ program file.cpp returns `True` when its input is the content of `file.cpp` itself.
Otherwise, output <u>`False`</u>.

Still we don't know how to show that there is no C++ program for **Problem-B**.

Reductions....

Problem-B

Input: One files: A C++ program, denoted by `file.cpp`.

Task: Output <u>`True`</u>, if the C++ program file.cpp returns `True` when its input is the content of `file.cpp` itself.
Otherwise, output <u>`False`</u>.

Still we don't know how to show that there is no C++ program for **Problem-B**.

So, we consider the following problem:

Problem-C

Input: One files: A C++ program, denoted by `file.cpp`.

Task: Output <u>`False`</u>, if the C++ program file.cpp returns `True` when its input is the content of `file.cpp` itself.
Otherwise, output <u>`True`</u>.

Reductions....

Problem-B

Input: One files: A C++ program, denoted by `file.cpp`.
Task: Output True, if the C++ program file.cpp returns `True` when its input is the content of `file.cpp` itself.
Otherwise, output False.

Still we don't know how to show that there is no C++ program for **Problem-B**.

So, we consider the following problem:

Problem-C

Input: One files: A C++ program, denoted by `file.cpp`.
Task: Output False, if the C++ program file.cpp returns `True` when its input is the content of `file.cpp` itself.
Otherwise, output True.

Note that the output of **Problem-B** is just the complement of the output of **Problem-C**.

Reductions....

> **Problem-B**
>
> **Input:** One files: A C++ program, denoted by `file.cpp`.
> **Task:** Output <u>`True`</u>, if the C++ program file.cpp returns `True` when its input is the content of `file.cpp` itself.
> Otherwise, output <u>`False`</u>.

Still we don't know how to show that there is no C++ program for **Problem-B**.

So, we consider the following problem:

> **Problem-C**
>
> **Input:** One files: A C++ program, denoted by `file.cpp`.
> **Task:** Output <u>`False`</u>, if the C++ program file.cpp returns `True` when its input is the content of `file.cpp` itself.
> Otherwise, output <u>`True`</u>.

Note that the output of **Problem-B** is just the complement of the output of **Problem-C**.

There is a C++ program for **Problem-B**, if and only if there is a C++ program for **Problem-C**.

Mathematical proof that there is no C++ program for Problem-C

Problem-C

Input: One files: A C++ program, denoted by `input.cpp`.

Task: Output <u>`False`</u>, if the C++ program input.cpp returns `True` when its input is the content of `input.cpp` itself.

Otherwise, output <u>`True`</u>.

Mathematical proof that there is no C++ program for Problem-C

Problem-C

Input: One files: A C++ program, denoted by `input.cpp`.

Task: Output `False`, if the C++ program input.cpp returns `True` when its input is the content of `input.cpp` itself.
Otherwise, output `True`.

The proof is by contradiction. Suppose there is a C++ program for **Problem-C**, which we denote by `myprog.cpp`.

Mathematical proof that there is no C++ program for Problem-C

Problem-C

Input: One files: A C++ program, denoted by `input.cpp`.

Task: Output <u>`False`</u>, if the C++ program input.cpp returns `True` when its input is the content of `input.cpp` itself.
Otherwise, output <u>`True`</u>.

The proof is by contradiction. Suppose there is a C++ program for **Problem-C**, which we denote by `myprog.cpp`.

Now we run `myprog.cpp` with input `myprog.cpp` itself. There are two cases.

Mathematical proof that there is no C++ program for Problem-C

Problem-C

Input: One files: A C++ program, denoted by `input.cpp`.
Task: Output <u>`False`</u>, if the C++ program input.cpp returns `True` when its input is the content of `input.cpp` itself.
Otherwise, output <u>`True`</u>.

The proof is by contradiction. Suppose there is a C++ program for **Problem-C**, which we denote by `myprog.cpp`.

Now we run `myprog.cpp` with input `myprog.cpp` itself. There are two cases.

- The output is `True`.

Mathematical proof that there is no C++ program for Problem-C

Problem-C

Input: One files: A C++ program, denoted by `input.cpp`.
Task: Output False, if the C++ program input.cpp returns `True` when its input is the content of `input.cpp` itself.
Otherwise, output True.

The proof is by contradiction. Suppose there is a C++ program for **Problem-C**, which we denote by `myprog.cpp`.

Now we run `myprog.cpp` with input `myprog.cpp` itself. There are two cases.

- The output is `True`.
  Since `myprog.cpp` is a program for **Problem-C**, by definition of **Problem-C**, `myprog.cpp` does not return `True` on `myprog.cpp` itself. A contradiction.

Mathematical proof that there is no C++ program for Problem-C

Problem-C

Input: One files: A C++ program, denoted by `input.cpp`.
Task: Output False, if the C++ program input.cpp returns `True` when its input is the content of `input.cpp` itself.
Otherwise, output True.

The proof is by contradiction. Suppose there is a C++ program for **Problem-C**, which we denote by `myprog.cpp`.

Now we run `myprog.cpp` with input `myprog.cpp` itself. There are two cases.

- The output is `True`.
  Since `myprog.cpp` is a program for **Problem-C**, by definition of **Problem-C**, `myprog.cpp` does not return `True` on `myprog.cpp` itself. A contradiction.
- The output is `False`.

Mathematical proof that there is no C++ program for Problem-C

Problem-C

Input: One files: A C++ program, denoted by `input.cpp`.
Task: Output False, if the C++ program input.cpp returns `True` when its input is the content of `input.cpp` itself.
Otherwise, output True.

The proof is by contradiction. Suppose there is a C++ program for **Problem-C**, which we denote by `myprog.cpp`.

Now we run `myprog.cpp` with input `myprog.cpp` itself. There are two cases.

- The output is `True`.
  Since `myprog.cpp` is a program for **Problem-C**, by definition of **Problem-C**, `myprog.cpp` does not return `True` on `myprog.cpp` itself. A contradiction.
- The output is `False`.
  Since `myprog.cpp` is a program for **Problem-C**, by definition of **Problem-C**, `myprog.cpp` returns `True` on `myprog.cpp` itself. A contradiction.

Mathematical proof that there is no C++ program for Problem-C

Problem-C

Input: One files: A C++ program, denoted by `input.cpp`.
Task: Output False, if the C++ program input.cpp returns `True` when its input is the content of `input.cpp` itself.
Otherwise, output True.

The proof is by contradiction. Suppose there is a C++ program for **Problem-C**, which we denote by `myprog.cpp`.

Now we run `myprog.cpp` with input `myprog.cpp` itself. There are two cases.

- The output is `True`.
  Since `myprog.cpp` is a program for **Problem-C**, by definition of **Problem-C**, `myprog.cpp` does not return `True` on `myprog.cpp` itself. A contradiction.
- The output is `False`.
  Since `myprog.cpp` is a program for **Problem-C**, by definition of **Problem-C**, `myprog.cpp` returns `True` on `myprog.cpp` itself. A contradiction.

Therefore, there is no such C++ program `myprog.cpp` for **Problem-C**.

Proof of Theorem 0.1

Proof of Theorem 0.1

Since there is no C++ program for **Problem-C**, there is no C++ program for **Problem-B**.

Proof of Theorem 0.1

Since there is no C++ program for **Problem-C**, there is no C++ program for **Problem-B**.

Since there is no C++ program for **Problem-B**, there is no C++ program for **Problem-A**.

Proof of Theorem 0.1

Since there is no C++ program for **Problem-C**, there is no C++ program for **Problem-B**.

Since there is no C++ program for **Problem-B**, there is no C++ program for **Problem-A**.

Thus, we complete the proof of Theorem 0.1.

Theorem 0.1

There is no C++ program for **Problem-A***.*

Proof of Theorem 0.1

Since there is no C++ program for **Problem-C**, there is no C++ program for **Problem-B**.

Since there is no C++ program for **Problem-B**, there is no C++ program for **Problem-A**.

Thus, we complete the proof of Theorem 0.1.

Theorem 0.1

There is no C++ program for **Problem-A**.

Problem-A is usually known as the "Halting" problem.

We will come back to it again when we discuss Turing machines in the next few months.

Table of contents

1. Introduction

2. Some words about mathematical proofs

3. The halting problem in C++

4. The notion of alphabets and languages

5. Concluding remarks

Language theoretic terminology

In this course we assume familiarity with basic terminology from discrete mathematics, especially set-theoretic terminology.

See Appendix A in Note 0 for some that we will often use.

Language theoretic terminology

In this course we assume familiarity with basic terminology from discrete mathematics, especially set-theoretic terminology.

See Appendix A in Note 0 for some that we will often use.

In addition, we will use the following terminology.

(Def.) An *alphabet* is a finite set of symbols.

We usually use the symbol Σ to denote an alphabet.

Language theoretic terminology

In this course we assume familiarity with basic terminology from discrete mathematics, especially set-theoretic terminology.

See Appendix A in Note 0 for some that we will often use.

In addition, we will use the following terminology.

(Def.) An *alphabet* is a finite set of symbols.

We usually use the symbol Σ to denote an alphabet.

Often, $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b\}$.

Language theoretic terminology

In this course we assume familiarity with basic terminology from discrete mathematics, especially set-theoretic terminology.

See Appendix A in Note 0 for some that we will often use.

In addition, we will use the following terminology.

(Def.) An *alphabet* is a finite set of symbols.

We usually use the symbol Σ to denote an alphabet.

Often, $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b\}$.

In standard computers, the alphabet is the set of all ASCII codes.

Language theoretic terminology

In this course we assume familiarity with basic terminology from discrete mathematics, especially set-theoretic terminology.

See Appendix A in Note 0 for some that we will often use.

In addition, we will use the following terminology.

(Def.) An *alphabet* is a finite set of symbols.

We usually use the symbol Σ to denote an alphabet.

Often, $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b\}$.

In standard computers, the alphabet is the set of all ASCII codes.

In memory level, the alphabet is $\{0, 1\}$.

(Finite) string/word

(Def.) A (finite) string/word over Σ is a finite sequence of symbols from Σ.

(Finite) string/word

(Def.) A (finite) string/word over Σ is a finite sequence of symbols from Σ.

For example, aaa, 0010, 0, 1111 are (finite) string

$00000\ldots$ (of infinite length) is not a string.

(Finite) string/word

(Def.) A (finite) string/word over Σ is a finite sequence of symbols from Σ.

For example, aaa, 0010, 0, 1111 are (finite) string

$00000\ldots$ (of infinite length) is not a string.

We usually write $w = a_1 \ldots a_n$ to denote a string whose symbol in position i is a_i.

(Finite) string/word

(Def.) A (finite) string/word over Σ is a finite sequence of symbols from Σ.

For example, aaa, 0010, 0, 1111 are (finite) string

$00000\ldots$ (of infinite length) is not a string.

We usually write $w = a_1 \ldots a_n$ to denote a string whose symbol in position i is a_i.

(Def.) The length of w is denoted by $|w|$.

(Finite) string/word

(Def.) A (finite) string/word over Σ is a finite sequence of symbols from Σ.

For example, aaa, 0010, 0, 1111 are (finite) string

$00000\ldots$ (of infinite length) is not a string.

We usually write $w = a_1 \ldots a_n$ to denote a string whose symbol in position i is a_i.

(Def.) The length of w is denoted by $|w|$.

For example, $|aaa| = 3$ and $|0| = 1$.

(Finite) string/word

(Def.) A (finite) string/word over Σ is a finite sequence of symbols from Σ.

For example, aaa, 0010, 0, 1111 are (finite) string

$00000\ldots$ (of infinite length) is not a string.

We usually write $w = a_1 \ldots a_n$ to denote a string whose symbol in position i is a_i.

(Def.) The length of w is denoted by $|w|$.

For example, $|aaa| = 3$ and $|0| = 1$.

(Def.) We write ε to denote the *empty string/word*, i.e., the word of length 0.

Languages

Let Σ be an alphabet.

Languages

Let Σ be an alphabet.

(Def.) For an integer $n \geqslant 0$, Σ^n denotes the set of all the words over Σ of length n.

Languages

Let Σ be an alphabet.

(Def.) For an integer $n \geqslant 0$, Σ^n denotes the set of all the words over Σ of length n.

(Def.) Σ^* denotes the set of all *finite* words over Σ, i.e., $\Sigma^* = \bigcup_{n \geqslant 0} \Sigma^n$.

Languages

Let Σ be an alphabet.

(Def.) For an integer $n \geqslant 0$, Σ^n denotes the set of all the words over Σ of length n.

(Def.) Σ^* denotes the set of all *finite* words over Σ, i.e., $\Sigma^* = \bigcup_{n \geqslant 0} \Sigma^n$.

(Def.) A *language* L over Σ is a subset of Σ^*.

Some examples of languages over $\Sigma = \{0, 1\}$

Some examples of languages over $\Sigma = \{0, 1\}$

- $\emptyset$ is a language over Σ.

Some examples of languages over $\Sigma = \{0, 1\}$

- $\emptyset$ is a language over Σ.
- Σ^* is a language over Σ.

Some examples of languages over $\Sigma = \{0, 1\}$

- $\emptyset$ is a language over Σ.
- Σ^* is a language over Σ.
- $\{w \in \Sigma^* | \text{the length of } w \text{ is } \leqslant 4\}$ is a language over Σ.

Some examples of languages over $\Sigma = \{0, 1\}$

- $\emptyset$ is a language over Σ.
- Σ^* is a language over Σ.
- $\{w \in \Sigma^* | \text{the length of } w \text{ is } \leqslant 4\}$ is a language over Σ.
- $\{w \in \Sigma^* | w \text{ does not contain } 0\}$ is a language over Σ.

Some examples of languages over $\Sigma = \{0, 1\}$

- $\emptyset$ is a language over Σ.
- Σ^* is a language over Σ.
- $\{w \in \Sigma^* | \text{the length of } w \text{ is } \leqslant 4\}$ is a language over Σ.
- $\{w \in \Sigma^* | w \text{ does not contain } 0\}$ is a language over Σ.
- $\{w \in \Sigma^* | \text{the length of } w \text{ is a prime number}\}$ is a language over Σ.

Table of contents

1. Introduction
2. Some words about mathematical proofs
3. The halting problem in C++
4. The notion of alphabets and languages
5. Concluding remarks

Concluding remarks

Concluding remarks

1. Introduction.

   https://www.csie.ntu.edu.tw/~tonytan/teaching/2021a-aut/2021a-aut.html

Concluding remarks

1. Introduction.

   `https://www.csie.ntu.edu.tw/~tonytan/teaching/2021a-aut/2021a-aut.html`

2. Some words about mathematical proofs.

   See Appendix A and B in Note 0 for standard terminology that we will use.

Concluding remarks

1. Introduction.

   `https://www.csie.ntu.edu.tw/~tonytan/teaching/2021a-aut/2021a-aut.html`

2. Some words about mathematical proofs.

   See Appendix A and B in Note 0 for standard terminology that we will use.

3. Halting problem for C++ (or any programming language).

Concluding remarks

1. Introduction.

   `https://www.csie.ntu.edu.tw/~tonytan/teaching/2021a-aut/2021a-aut.html`

2. Some words about mathematical proofs.

   See Appendix A and B in Note 0 for standard terminology that we will use.

3. Halting problem for C++ (or any programming language).

4. The notion of alphabets and languages.

   It will be used throughout the course.

End of Lesson 0