"Honestly? I preferred when we didn't talk about the elephant."

Shannon Wheeler, 2011
Student survey

forms.gle/67ESyzaW3gaGHgRV6
What’s a computer?
“Hey, whatcha doing on your computer?”

“What’s a computer?”
Seriously, what is a computer?
314
+  159
-----

🤔
314
+ 159
___
3
\[
\begin{array}{c}
1 \\
314 \\
+ \ 159 \\
\hline \\
73
\end{array}
\]
\[
\begin{array}{c}
1 \\
314 \\
+ 159 \\
\hline
473
\end{array}
\]
A small suan pan (Chinese abacus)

Photo from the Computer History Museum
Hand-cranked Curta calculator, c. 1950
Photo from the Computer History Museum
Some kinds of computers have more computational power than others.

We can abstract devices of the “same kind” to produce *models* of computers and ask what kinds of problems can be solved under a particular model.
One of the most remarkable things about computers is that their essential nature transcends technology.

Present-day computers are built out of transistors and wires.

They could be built, according to the same principles, from valves and water pipes or from sticks and strings.
Mechanical implementation of the or function

W. Daniel Hillis,
The Pattern on the Stone, 1998
Hydraulic implementation of the **or** function

W. Daniel Hillis,
*The Pattern on the Stone*,
1998
Why do we need theory?
Theory shows the elegant side of computers

We usually think of computers as complicated machines.

The best computer designs and applications are conceived with elegance in mind.

A theoretical course can heighten your aesthetic sense and help you build more elegant systems.
Theory expands your mind

Computer technology changes quickly.

Studying theory enables you to understand the underlying models of all computation, not just technical details that become outdated in a few years.

Studying theory trains you in abilities with lasting value:

- Think and express yourself clearly and precisely.
- Solve problems – and know when you haven’t solved a problem.
Theory is relevant to practice

Provides conceptual tools that practitioners use in computer engineering

Design a new programming language for a special application – need (context-free) grammars!

String searching and pattern matching – use finite automata and regular expressions!
When developing solutions to real problems, we often confront the limitations of what software can do:

*Undecidable things* – no program whatever can do it

*Intractable things* – there are programs, but no fast programs

Theory gives you the tools.
He who loves practice without theory is like the sailor who boards ship without a rudder and compass and never knows where he may cast.

Leonardo da Vinci, “Prolegomena and General Introduction to the Book on Painting”
Theoretical computer science has many fascinating big ideas, but also many small (and sometimes dull) details.

The more you learn, the more interesting it becomes.

Our goal: Be exposed to the exciting aspects of computer theory without getting bogged down in drudgery.
How could we talk about computation?
Automata, computability, and complexity

Linked by the question: *What are the fundamental capabilities and limitations of computers?*

Each area interprets the question differently:

*Automata theory:* Definitions and properties of mathematical models of computation.

*Computability:* Is a given problem solvable or unsolvable?

*Complexity:* Is a given problem easy or hard?
Automata theory

We start with *automata theory*:

Theories of computability and complexity require a *precise definition of a computer*.

Automata theory allows practice with formal definitions of computation as it introduces concepts relevant to other, non-theoretical areas of computer science.
The central idea in the theory of computation is that of a *universal computer*, a computer powerful enough to simulate any other computing device.

Most computers we encounter in everyday life are universal computers.

With the right software – and enough time and memory – they can simulate any other type of computer…
Universal computers
Replacing a bad tube meant checking among ENIAC’s 19,000 possibilities.
The idea of a universal computer was recognized and described in 1937 by Alan Turing.¹

He called it a “universal machine” since at the time, “computer” still meant “a person who performs computations”.

¹ Poor Alonzo Church is a footnote. Where’s his movie?
The idea of a universal computer was recognized and described in 1937 by Alan Turing.\textsuperscript{1}

He called it a “universal machine” since at the time, “computer” still meant “a person who performs computations”.

\textsuperscript{1} Poor Alonzo Church is a footnote. Where’s his movie?
The central idea in the theory of computation is that of a *universal computer*, a computer powerful enough to simulate any other computing device.

Most computers we encounter in everyday life are universal computers.

With the right software – and enough time and memory – they can simulate any other type of computer… *or – as far as we know – any other device at all that processes information.*
HYPOTHESIS: *Any* computing device – made of transistors, sticks and strings, or neurons – can be simulated by a universal computer.

This suggests that making a computer think like a brain is just a matter of programming it correctly!
While a universal computer can compute anything that can be computed any other computing device, there are some things that are just impossible to compute.
Questions for which we lack data

“What is the winning number in tomorrow’s lottery?”
Vaguely defined questions

“What is the meaning of life?”


“42!”
But there are also flawlessly defined computational problems that are impossible to solve.

We call these problems *noncomputable*. 
What exactly are the limits to what a computer can do?

We'll work to an answer of this over the semester!

This will take us through the philosophically interesting topics of nondeterminism, Turing machines, computability, and Gödel's incompleteness theorem.
Course overview

Study categories of languages and machines:

- *Regular languages* and *finite automata*
- *Context-free languages* and *pushdown automata*
- *Unrestricted languages* and *Turing machines*

Study solvability:

*The Halting Problem* and its ramifications
Course information
Prerequisites

CMPU 102: Data Structures and Algorithms

CMPU 145: Foundations of Computer Science
CMPU 240
Theory of Computation
Fall 2021

Tuesday & Thursday, 1:30–2:45 p.m.
Sanders Physics 309

Prof. Jonathan Gordon

SYLLABUS

<table>
<thead>
<tr>
<th></th>
<th>Tuesday</th>
<th>Thursday</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read Syllabus</td>
<td>Lecture 1</td>
</tr>
<tr>
<td></td>
<td>Read §0</td>
<td>Theory of computation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lecture 2</td>
</tr>
</tbody>
</table>

2 Finite automata

<table>
<thead>
<tr>
<th></th>
<th>Sep. 7</th>
<th>Sep. 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lecture 3</td>
<td>Lecture 4</td>
</tr>
</tbody>
</table>

[cs.vassar.edu/~cs240]
On the course website will be links to the other two sites we use:

**Campuswire**

- Use for general discussion about the course content
- Use for all questions about the course
  
  - You can post anonymously.
  - You can send private questions to me.

**Gradescope**

- Use for submitting assignments and receiving feedback
Grading

- Assignments: 35%
  - 8–10 homework assignments
Grading

- Assignments: 35%
- Exam 1: 20%
- Exam 1: 20%
- Exam 1: 20%
Grading

- Assignments: 35%
- Exam 1: 20%
- Exam 2: 20%

Exam 2
Grading

- Assignments: 25%
- Exam 1: 20%
- Exam 2: 20%
- Exam 3: 35%

Regularly scheduled final Exam 3
"THIS IS THE BEST BOOK ON COMPUTERS I HAVE EVER READ."
—PETER THOMAS, NEW SCIENTIST

THE PATTERN ON THE STONE

THE SIMPLE IDEAS THAT MAKE COMPUTERS WORK

W. DANIEL HILLIS
Hillis’s Connection Machine CM-2a

Photo by Steve Grohe for Thinking Machines Corporation
Hillis’s Connection Machine CM-5 in Jurassic Park
Syllabus

Read for more details on the course.

There will be an additional handout on how assignments will work.
Language theory and computation
Simplifying computation

A string \rightarrow \ldots \rightarrow Yes \rightarrow No
"01001110101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
A string → ... → Yes, No
\[4 \text{ sqrt } 2\]
"4, 2" is-sqrt True
\[ f(x) = y \]

\[ f'(x, y) = \text{Yes} \]

\[ \sqrt{4} = 2 \]

\[ \text{is-sqrt}(4, 2) = \text{Yes} \]

\[ \text{is-sqrt}(4, 3) = \text{No} \]
Simplifying computation

A string → ... → Yes, No
Strings

A string is a sequence of characters/symbols.
Strings

A *string* is a sequence of characters/symbols.

In a programming language, a string is written with quotation marks, e.g., "*sleepy otters*".

  Punctuation goes outside the quotation marks because we’re not animals.
Strings

A *string* is a sequence of characters/symbols.

In a programming language, a string is written with quotation marks, e.g., "*sleepy otters*".

  Punctuation goes outside the quotation marks because we’re not animals.

In language theory, we omit the quotation marks and use visible characters for any spaces, e.g., `sleepy otters`. 
Strings

If you’re programming, what’s """"? 
Strings

If you’re programming, what’s ""?  

It’s a string of length 0. (In fact, it’s the only string of length 0!)
Strings

If you’re programming, what’s ""? It’s a string of length 0. (In fact, it’s the only string of length 0!)

We call it the empty string.
Strings

If you’re programming, what’s """"? It’s a string of length 0. (In fact, it’s the only string of length 0!)

We call it the empty string.

Because we don’t use quotation marks in theory, we write it as epsilon, $\varepsilon$. 
A set of strings is called a *language*.
A set is an unordered collection of 0 or more objects of any type, e.g.,

Ø

{0}

{0, 1}

\(\mathbb{N}\)

\(\mathbb{N}_0\)
A set is an unordered collection of 0 or more objects of any type, e.g.,

- $\emptyset$ 0 objects – the empty set
- \{0\} 1 object – the set containing the number 0
- \{0, 1\} 2 objects – the set containing the numbers 0 and 1
- $\mathbb{N}$ infinite objects – the set of all natural numbers
- $\mathbb{N}_0$ infinite objects – the set of all natural numbers including 0
For a set to be a language, it can’t have any elements except for strings.

∅

{ε}

{a}

{a, b}
For a set to be a language, it can’t have any elements except for strings.

- $\emptyset$ (0 strings – the *empty language*)
- $\{\varepsilon\}$ (1 string – the language containing the empty string)
- $\{a\}$ (1 string – the language containing the string $a$)
- $\{a, b\}$ (2 strings – the language containing the strings $a$ and $b$)
What, then, is the English language?
What, then, is the English language?

The C programming language?
The set of all binary strings consisting of some number of 0s followed by an equal number of 1s?
The set of all binary strings consisting of some number of 0s followed by an equal number of 1s?

\{ \varepsilon, 01, 0011, 000111, \ldots \}
A language can be finite, i.e., only contain a fixed number of strings, even if that number is large.

A language can be infinite, i.e., contain an unbounded number of strings.
What does this have to do with computation?
“The set (language) of all computer programs”

“The set (language) of all problems to solve”
Thus, for any nondeterministic Turing machine \( M \) that runs in some polynomial time \( P(n) \), we can devise an algorithm that takes an input \( w \) of length \( n \) and produces \( E_{M,w} \). The running time is \( O(P(n)) \) on a multitape deterministic Turing machine and...

WTF, man. I just wanted to learn how to program video games.
Acknowledgments

This lecture incorporates material from:

David Chiang, University of Notre Dame
W. Daniel Hillis, The Pattern on the Stone
John Hopcroft, Cornell University
Nancy Ide, Vassar College
Keith Schwarz, Stanford University
Michael Sipser, Introduction to the Theory of Computation