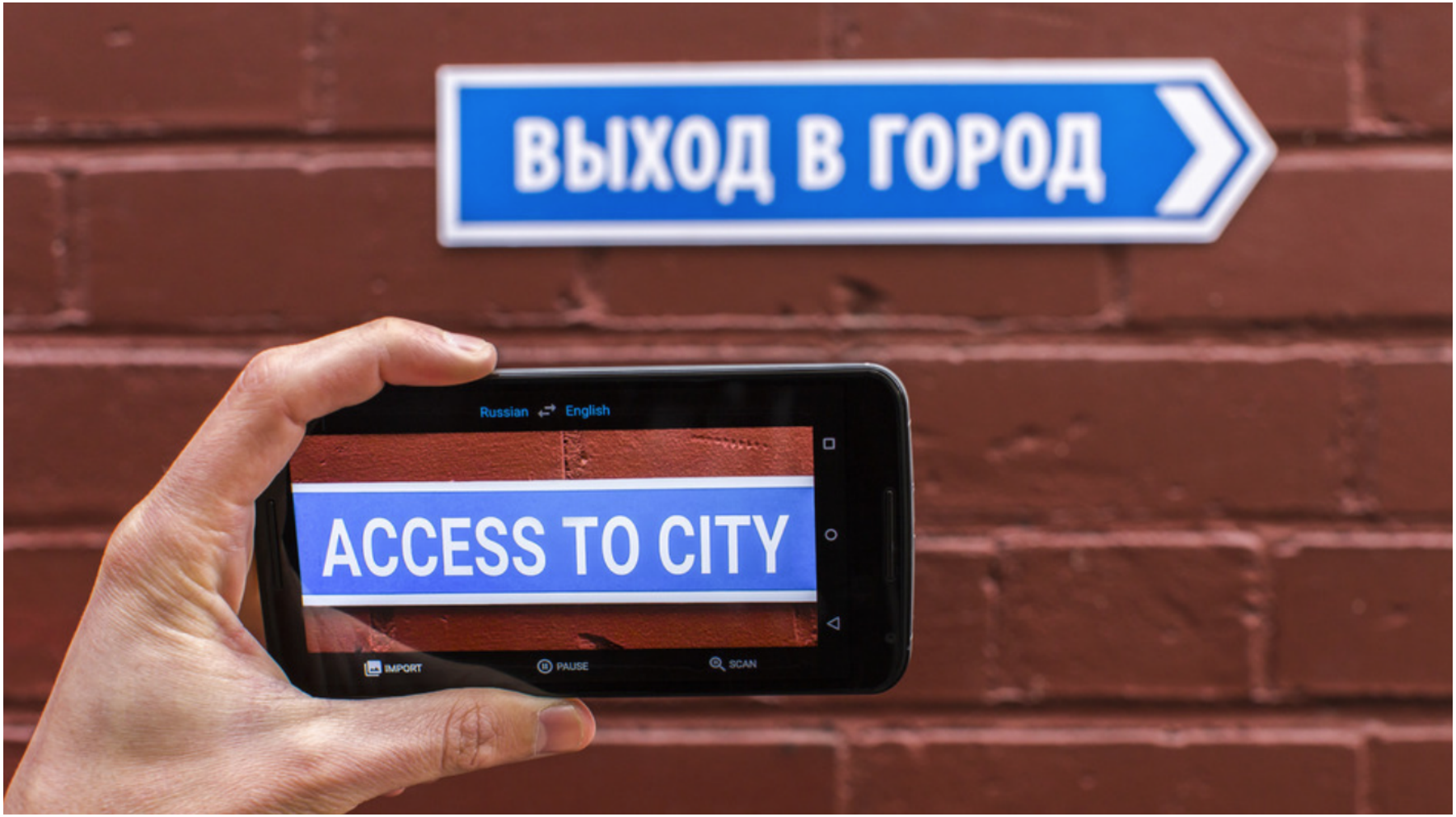
COSE312: Compilers

Lecture 20 — The Origin of Computer Revolution

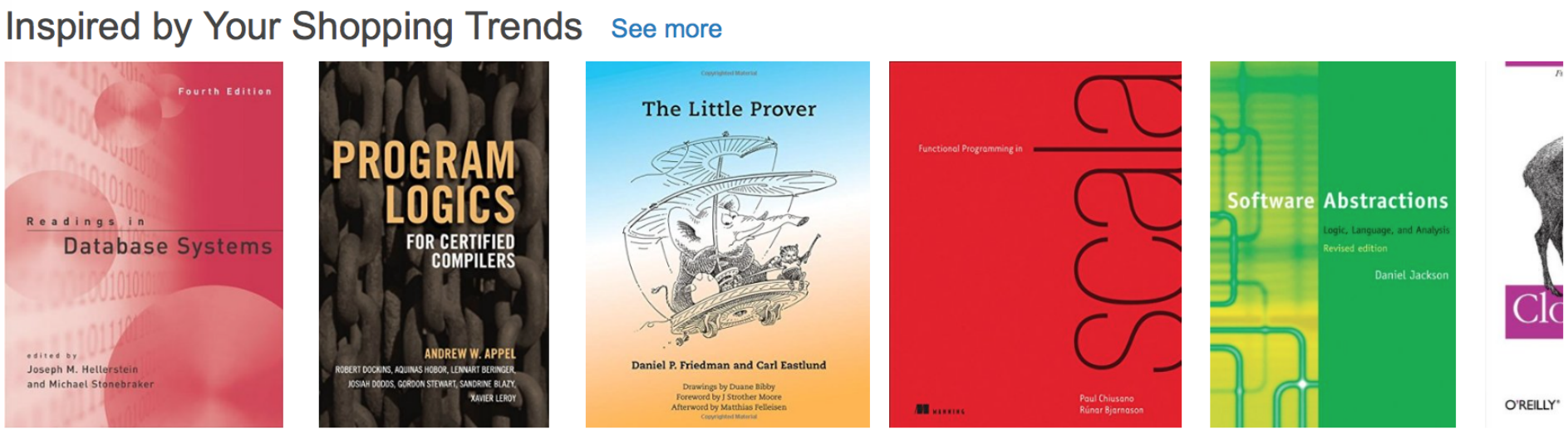
Hakjoo Oh
2015 Fall

Computer Revolution

Computer Revolution

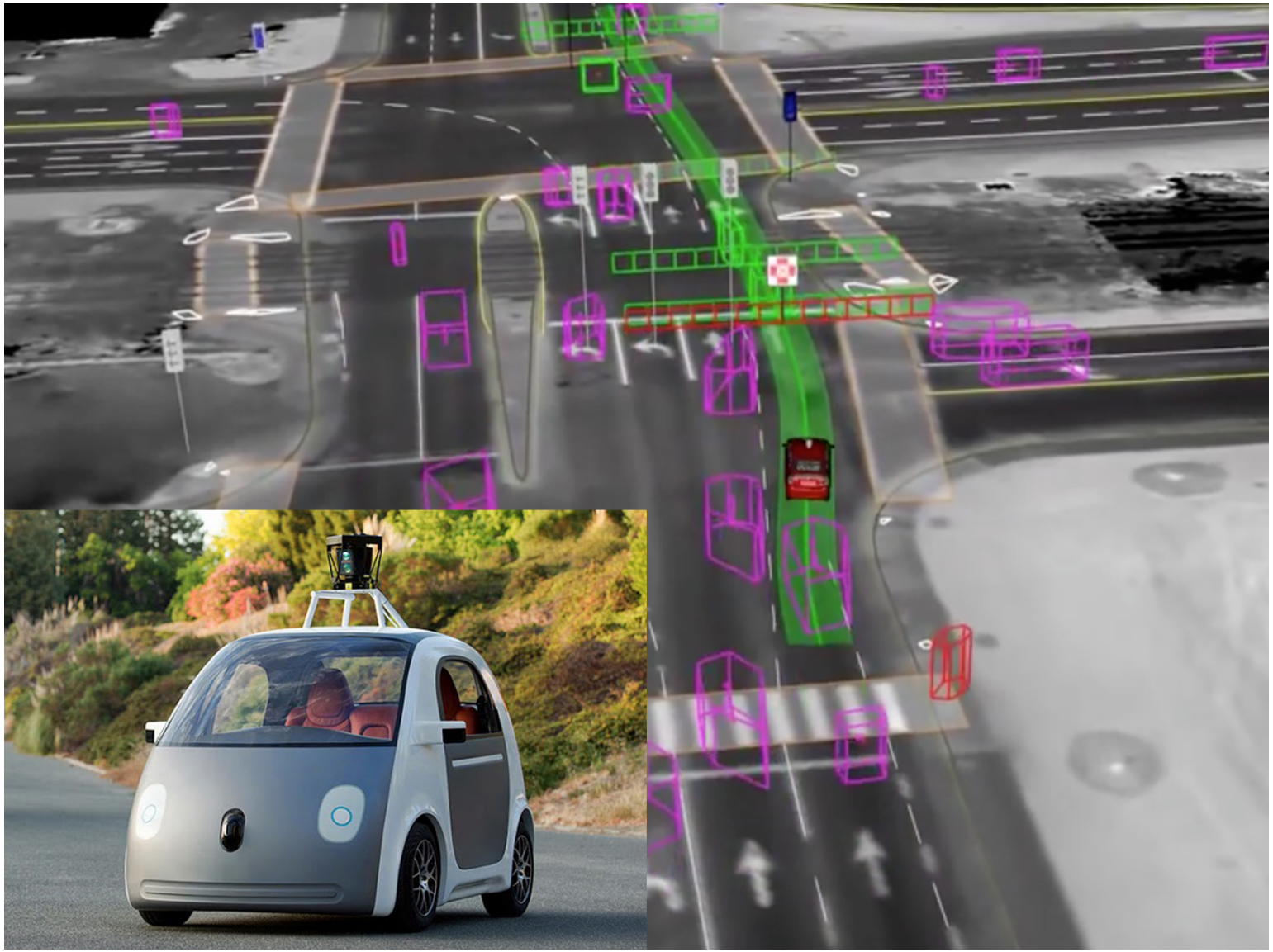


Computer Revolution



Computer Revolution

Computer Revolution



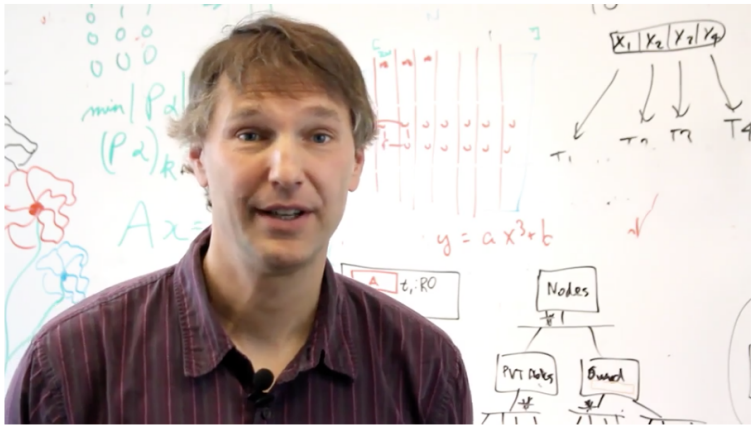
Computer Revolution

coursera Catalog Search catalog Institutions HO

Stanford
Compilers

This course will discuss the major ideas used today in the implementation of programming language compilers. You will learn how a program written in a high-level language designed for humans is systematically translated into a program written in low-level assembly more suited to machines!

Preview Lectures



About the Course

This course will discuss the major ideas used today in the implementation of programming language compilers, including lexical analysis, parsing, syntax-directed translation, abstract syntax trees, types and type checking, intermediate languages, dataflow analysis, program optimization, code generation, and runtime systems. As a result, you will learn how a program written in a high-level language designed for humans is systematically translated into a program written in low-level assembly more suited to machines. Along the way we will also touch on how programming languages are designed, programming language semantics, and why there are so many different kinds of programming languages.

The course lectures will be presented in short videos. To help you master the material, there will be in-lecture questions to answer, quizzes, and two exams: a

Sessions

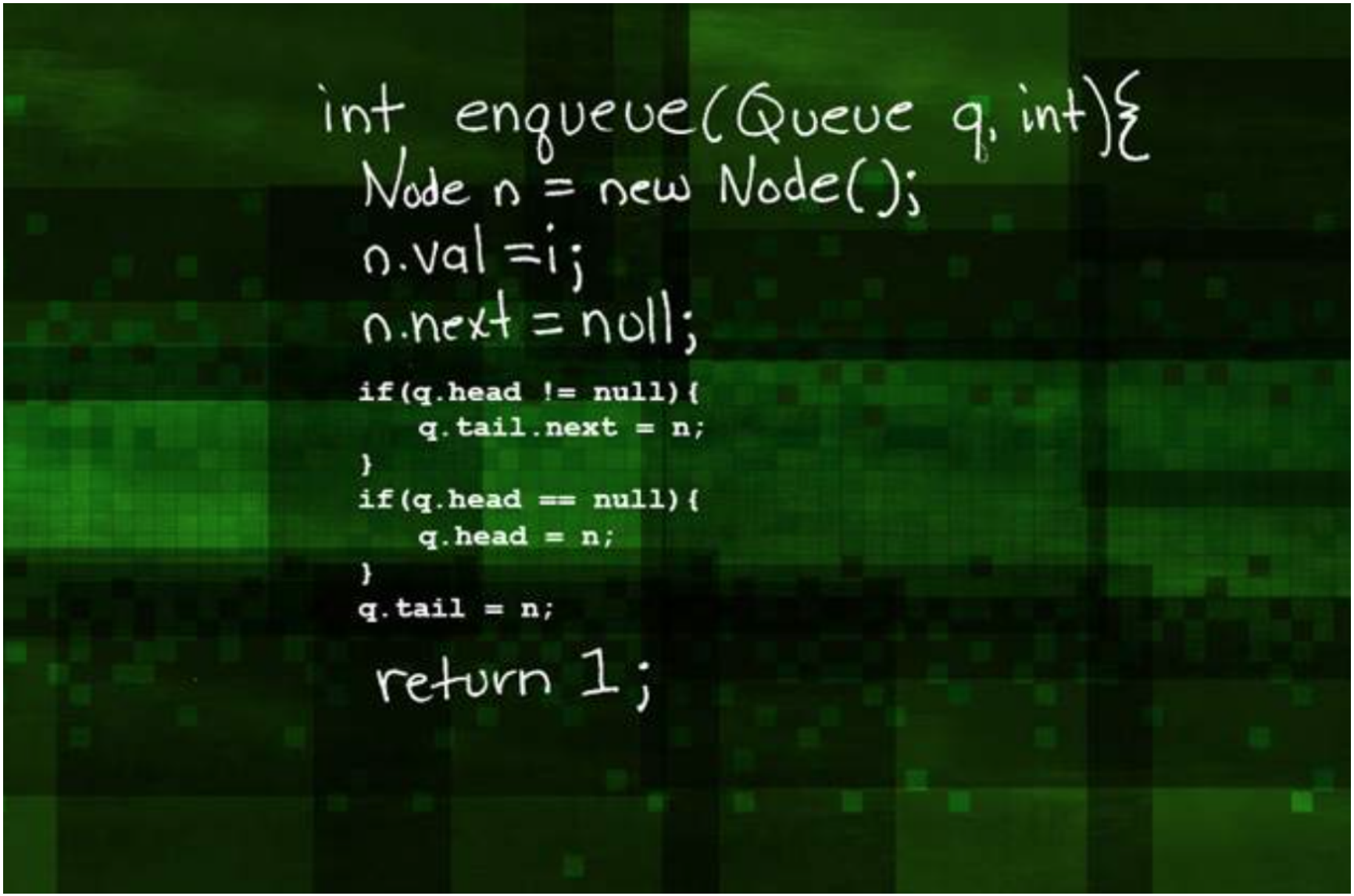
March 17, 2014 - June 2, 2014

Enroll

Course at a Glance

11 weeks of study
8-10 hours/week, 10-20 hours/week with programming assignments
English

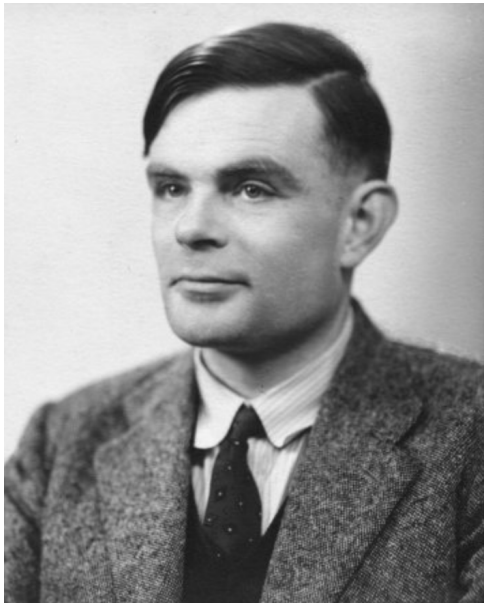
Computer Revolution



The Origin of Computer Revolution

The Origin of Computer Revolution

Church (1903-1995)



Turing (1912-1954)

- In 1936, Alonzo Church invented "Lambda calculus", the origin of programming languages.
- In 1936, Alan Turing invented "Turing machine", the origin of computers.

The Origin of Computer Revolution

- Alan J. Perlis, John McCarthy, Edsger W. Dijkstra, Donald E. Knuth, Dana S. Scott, John Backus, Robert W. Floyd, Kenneth E. Iverson, Tony Hoare, Stephen A. Cook, Niklaus Wirth, John Hopcroft, Robin Milner, Amir Pnueli, Ole-Johan Dahl, Peter Naur, Frances E. Allen, Alan Kay, Edmund M. Clarke, etc

The Origin of Computer Revolution

What is most surprising in Turing's work?

The concept of "universal machine".

The Origin of Computer Revolution

What is most surprising in Turing's work?

The concept of "universal machine".

- We can design a Turing machine that can simulate an arbitrary Turing machine on an arbitrary input.

The Origin of Computer Revolution

What is most surprising in Turing's work?

The concept of "universal machine".

- We can design a Turing machine that can simulate an arbitrary Turing machine on an arbitrary input.
  - The set of Turing machines is *closed* in power.

The Origin of Computer Revolution

What is most surprising in Turing's work?

The concept of "universal machine".

- We can design a Turing machine that can simulate an arbitrary Turing machine on an arbitrary input.
  - The set of Turing machines is *closed* in power.
- cf) The universal Turing machine is an interpreter:

  - An interpreter can simulate an arbitrary program for arbitrary input.
  - The interpreter is just another program.
- What if computers/programming languages are not universal?

The Origin of Computer Revolution

What is most surprising in Turing's work?

The concept of "universal machine".

- We can design a Turing machine that can simulate an arbitrary Turing machine on an arbitrary input.
  - The set of Turing machines is *closed* in power.
- cf) The universal Turing machine is an interpreter:

  - An interpreter can simulate an arbitrary program for arbitrary input.
  - The interpreter is just another program.
- What if computers/programming languages are not universal?
- Turing machine and lambda calculus appeared as byproducts of mathematicians's dream.

What Mathematicians Dreamed About

Leibniz (1646-1716)

Hillbert (1862-1943)

- Hillbert's Entscheidungsproblem (1928 @ICM):

  "Is there an algorithm to decide whether a given first-order statement is provable from the axioms using the inference rules?"

cf) Peano Arithmetic

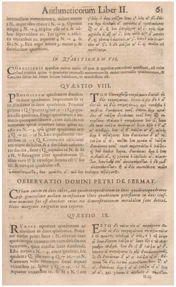
Vocabulary:

$$\Sigma = \{0, 1, +, \cdot, =\}$$

Axioms:

1. $\forall x. \neg(x + 1 = 0)$
2. $\forall x, y. x + 1 = y + 1 \implies x = y$
3. $F[0] \land (\forall x. F[x] \implies F[x + 1]) \implies \forall x. F[x]$
4. $\forall x. x + 0 = x$
5. $\forall x, y. x + (y + 1) = (x + y) + 1$
6. $\forall x. x \cdot 0 = 0$
7. $\forall x, y. x \cdot (y + 1) = x \cdot y + x$

cf) Peano Arithmetic


Arithmeticorum Liber II. 61

QVÆSTIO VIII.

OBSERVATIO DOMINI PETRI DE FERMAT.

QVÆSTIO IX.

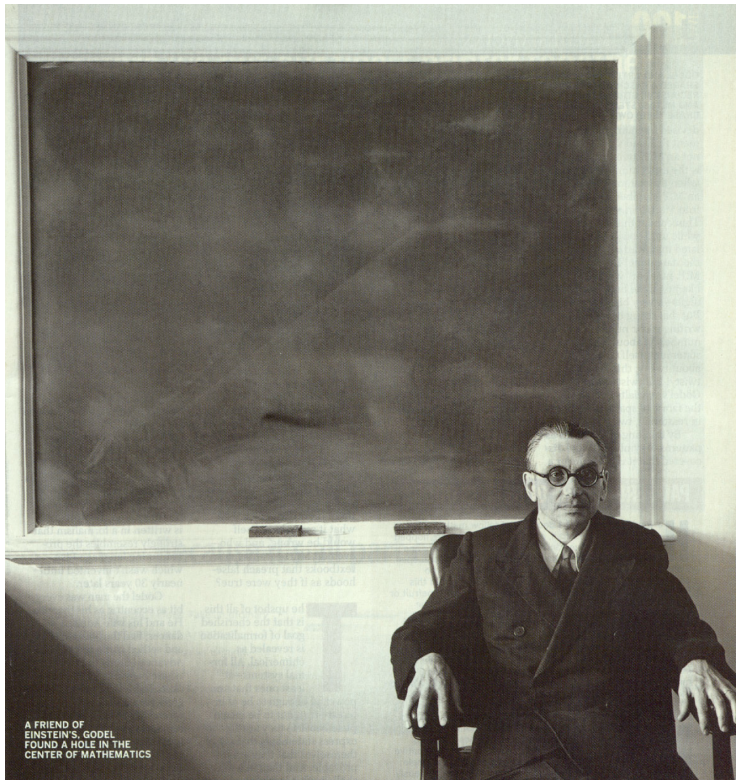
Theorem (Fermat's Last Theorem)

$$\forall x, y, z, n. x \neq 0 \land y \neq 0 \land z \neq 0 \land n > 2 \implies x^n + y^n \neq z^n$$

- Proposed by Fermat in 1637.
- Completely proved by Wiles in 1995.
- Can we automate the proof search? (Hillbert's program)

Gödel's Incompleteness Theorems (1931)

A complete and consistent set of axioms for all mathematics is impossible.



Kurt Gödel (1906-1978)

Direct Proofs by Turing and Church

In 1936, Alonzo Church and Alan Turing directly showed that a general solution to the decision problem is impossible.

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers π, e, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. These results

† Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", *Monatshefte Math. Phys.*, 38 (1931), 173–198.

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By ALONZO CHURCH.

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly the condition that the function f be effectively calculable is an essential part of the problem, since without it the problem becomes trivial.

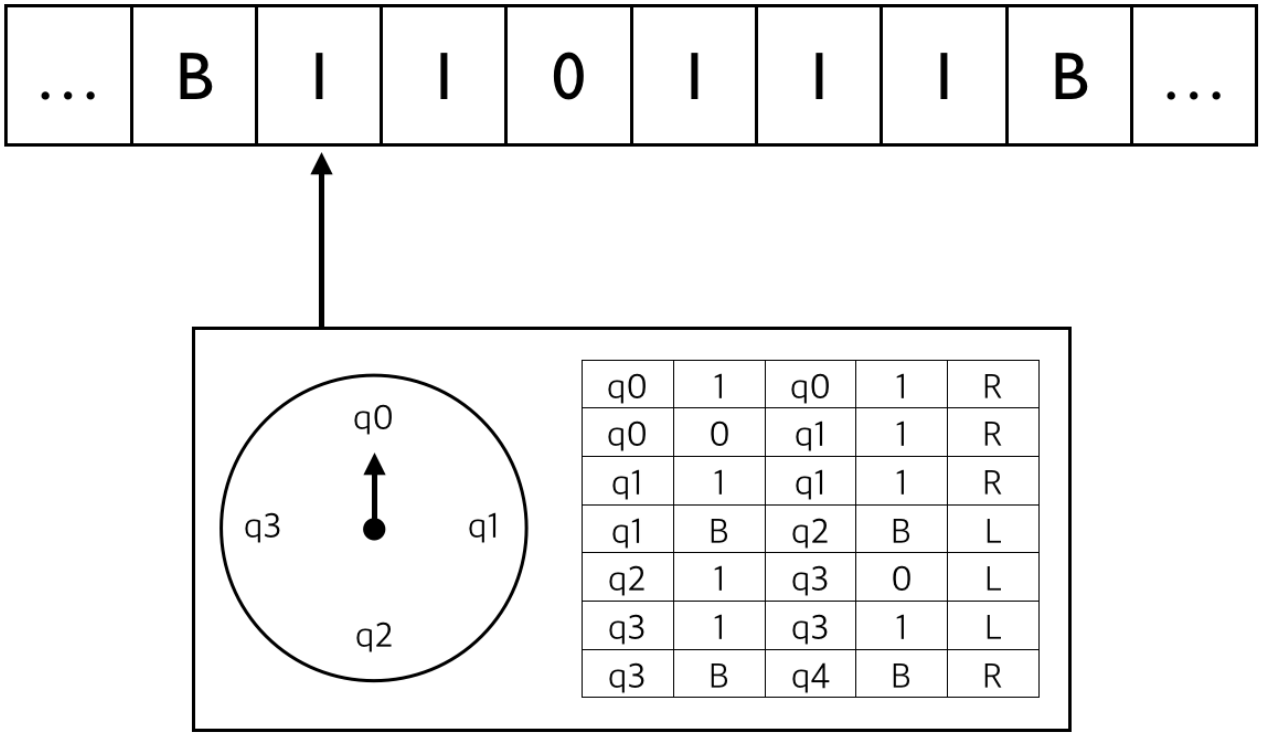
Another example of a problem of this class is, for instance, the problem of topology, to find a complete set of effectively calculable invariants of closed three-dimensional simplicial manifolds under homeomorphisms. This problem can be interpreted as a problem of elementary number theory in view of the fact that topological complexes are representable by matrices of incidence. In fact, as is well known, the property of a set of incidence matrices that it represent a closed three-dimensional manifold, and the property of two sets of incidence matrices that they represent homeomorphic complexes, can both be described in purely number-theoretic terms. If we enumerate, in a straightforward way, the sets of incidence matrices which represent closed three-dimensional manifolds, it will then be immediately provable that the problem under consideration (to find a complete set of effectively calculable invariants of closed three-dimensional manifolds) is equivalent to the problem, to find an effectively calculable function f of positive integers, such that $f(m, n)$ is equal to 2 if and only if the m-th set of incidence matrices and the n-th set of incidence matrices in the enumeration represent homeomorphic complexes.

Other examples will readily occur to the reader.

[1] Presented to the American Mathematical Society, April 19, 1935.

[2] The selection of the particular positive integer 2 instead of some other is, of course, accidental and non-essential.

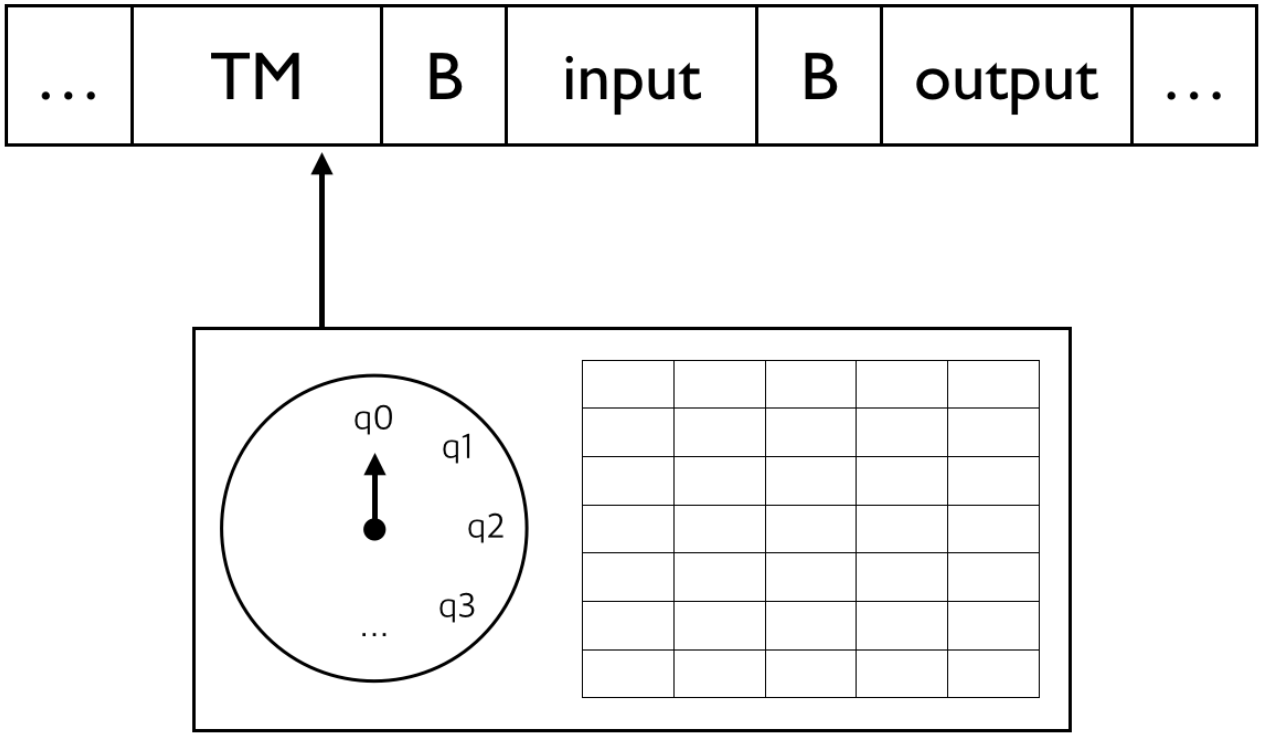
Turing's Definition of Computation



Examples:

- A machine to compute the sequence $\mathbf{010101\ldots}$
- A machine to compute the sequence $\mathbf{001011011101111011111\ldots}$

Universal Turing Machine



UTM is a Turing machine that can simulate an arbitrary Turing machine on an arbitrary input.

Turing's Proof

- Turing reduced the halting problem for Turing machines to the decision problem.
- H: the Turing machine that solves the halting problem
- A: the Turing machine that solves the decision problem
- $A \implies H$
- H is logically impossible:

	I_1	I_2	I_3	$\cdots$
M_1	1	1	0	$\cdots$
M_2	1	0	1	$\cdots$
M_3	1	0	1	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	

cf) Static Analysis

Turing showed that static analysis in general is impossible.

Church's Definition of Computation

- Lambda calculus:

$$\begin{array}{rcl} \boldsymbol{E} & \rightarrow & \boldsymbol{x} \\ & | & \boldsymbol{\lambda x.E} \\ & | & \boldsymbol{E\ E} \end{array}$$

- Computation (β-reduction):

$$(\boldsymbol{\lambda x.E})\ \boldsymbol{E'} \rightarrow [\boldsymbol{x} \mapsto \boldsymbol{E'}]\boldsymbol{E}$$

Example:

$$(\boldsymbol{\lambda x.(\lambda y.(x\ y))})\ \boldsymbol{z} \rightarrow \boldsymbol{\lambda y.(z\ y)}$$

The Power of Lambda Calculus

$$
\begin{array}{rcl}
\underline{\text{true}} & = & \lambda t.\lambda f.t \\
\underline{\text{false}} & = & \lambda t.\lambda f.f \\
\underline{\text{and}} & = & \lambda b.\lambda c.(b\ c\ \underline{\text{false}}) \\
\underline{\text{pair}} & = & \lambda f.\lambda s.\lambda b.b\ f\ s \\
\underline{\text{fst}} & = & \lambda p.p\ \underline{\text{true}} \\
\underline{\text{snd}} & = & \lambda p.p\ \underline{\text{false}} \\
\underline{\mathbf{0}} & = & \lambda s.\lambda z.z \\
\underline{\mathbf{1}} & = & \lambda s.\lambda z.(s\ z) \\
\underline{\mathbf{2}} & = & \lambda s.\lambda z.s\ (s\ z) \\
\underline{n} & = & \lambda s.\lambda z.s^n\ z \\
\underline{\text{plus}} & = & \lambda n.\lambda m.\lambda s.\lambda z.m\ s\ (n\ s\ z) \\
\underline{\text{mult}} & = & \lambda m.\lambda n.m\ (\underline{\text{plus}}\ n)\ c_0
\end{array}
$$

Impacts on Programming Languages

"Lambda" is everywhere:

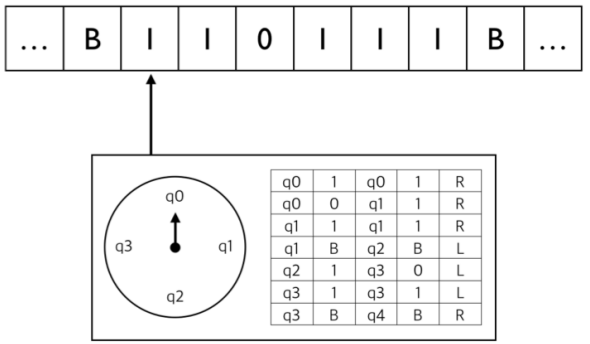
- Lisp, e.g., `(lambda (x) (* x x))`
- ML, e.g., `(fun x -> x * x)`
- JavaScript
- C#
- Java8
- C++11
- ...

Church's Proof

Church proved that there is no computable function which decides for two given lambda calculus expressions whether they are equivalent or not.

Church-Turing Thesis

$$
\begin{array}{rcl}
E & \rightarrow & x \\
 & | & \lambda x.E \\
 & | & E\ E
\end{array}
\quad = \quad
$$



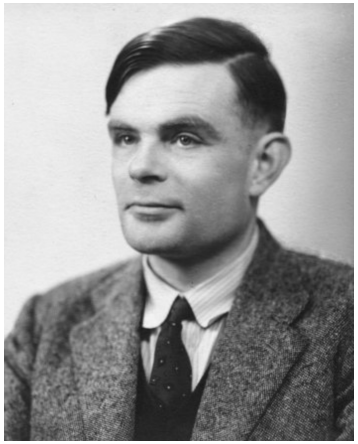
- Turing machine and lambda calculus are equally powerful.

Future of Computer Science?

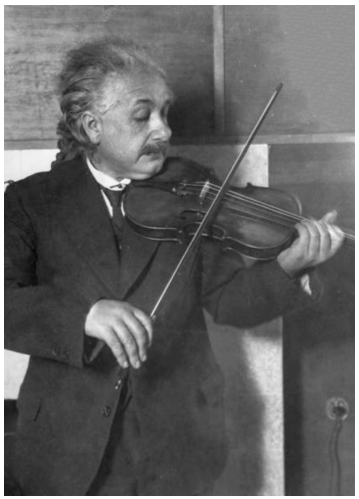
Future of Computer Science?

Newton (1642-1726)

vs.



Turing (1912-1954)



Einstein (1879-1955)

vs.